Web Application Exploitation 4

Fuzzing

Local File Inclusion/Remote File Inclusion

Polyglots

Race Conditions

DISCLAIMER

- All of the material in this school can be used for good (testing, research, educating), but also for bad
- We use our skills and knowledge responsibly and ethically
- We recommend you do the same
- We are not responsible for anything you do as a result of these lessons

Fuzzing

Throwing shit at the wall to see what sticks

Fuzzing is the concept of testing a range of values in a URL or request of any type to see the outcome. It’s not specific to web applications - all kinds of software can be fuzzed by giving it many inputs.

Fuzzing 2

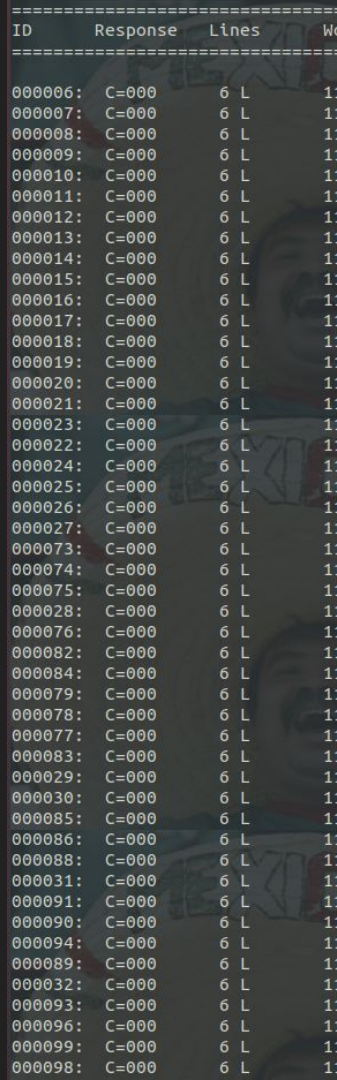
First, create the wordlist to attack:

```
~/pwnschool$ python -c "for i in range(0,2000): print(i)" > numbers
```

Then FUZZ it:

```
~/pwnschool$ wfuzz -w numbers 127.0.0.1:5000/FUZZ
```

Outcome:



```
=====================================================================
ID      Response   Lines      Word       Chars          Payload
=====================================================================

000006:  C=000      6 L       11 W       146 Ch         "5"
000007:  C=000      6 L       11 W       146 Ch         "6"
000008:  C=000      6 L       11 W       146 Ch         "7"
000009:  C=000      6 L       11 W       146 Ch         "8"
000010:  C=000      6 L       11 W       146 Ch         "9"
000011:  C=000      6 L       11 W       146 Ch         "10"
000012:  C=000      6 L       11 W       146 Ch         "11"
000013:  C=000      6 L       11 W       146 Ch         "12"
000014:  C=000      6 L       11 W       146 Ch         "13"
000015:  C=000      6 L       11 W       146 Ch         "14"
000016:  C=000      6 L       11 W       146 Ch         "15"
000017:  C=000      6 L       11 W       146 Ch         "16"
000018:  C=000      6 L       11 W       146 Ch         "17"
000019:  C=000      6 L       11 W       146 Ch         "18"
000020:  C=000      6 L       11 W       146 Ch         "19"
000021:  C=000      6 L       11 W       146 Ch         "20"
000023:  C=000      6 L       11 W       146 Ch         "22"
000022:  C=000      6 L       11 W       146 Ch         "21"
000024:  C=000      6 L       11 W       146 Ch         "23"
000025:  C=000      6 L       11 W       146 Ch         "24"
000026:  C=000      6 L       11 W       146 Ch         "25"
000027:  C=000      6 L       11 W       146 Ch         "26"
000073:  C=000      6 L       11 W       146 Ch         "72"
000074:  C=000      6 L       11 W       146 Ch         "73"
000075:  C=000      6 L       11 W       146 Ch         "74"
000028:  C=000      6 L       11 W       146 Ch         "27"
000076:  C=000      6 L       11 W       146 Ch         "75"
000082:  C=000      6 L       11 W       146 Ch         "81"
000084:  C=000      6 L       11 W       146 Ch         "83"
000079:  C=000      6 L       11 W       146 Ch         "78"
000078:  C=000      6 L       11 W       146 Ch         "77"
000077:  C=000      6 L       11 W       146 Ch         "76"
000083:  C=000      6 L       11 W       146 Ch         "82"
000029:  C=000      6 L       11 W       146 Ch         "28"
000030:  C=000      6 L       11 W       146 Ch         "29"
000085:  C=000      6 L       11 W       146 Ch         "84"
000086:  C=000      6 L       11 W       146 Ch         "85"
000088:  C=000      6 L       11 W       146 Ch         "87"
000031:  C=000      6 L       11 W       146 Ch         "30"
000091:  C=000      6 L       11 W       146 Ch         "90"
000090:  C=000      6 L       11 W       146 Ch         "89"
000094:  C=000      6 L       11 W       146 Ch         "93"
000089:  C=000      6 L       11 W       146 Ch         "88"
000032:  C=000      6 L       11 W       146 Ch         "31"
000093:  C=000      6 L       11 W       146 Ch         "92"
000096:  C=000      6 L       11 W       146 Ch         "95"
000099:  C=000      6 L       11 W       146 Ch         "98"
000098:  C=000      6 L       11 W       146 Ch         "97"
```

So, let’s clear it out:

Use --hh to filter any result with character length X:

```
:~/pwnschool$ wfuzz -w numbers --hh 146 127.0.0.1:5000/FUZZ
```

```
=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================

000002:  C=000      9 L      20 W       295 Ch      "1"
000133:  C=000      6 L      12 W       202 Ch      "132"
```

Remote File Inclusion

Remote File Inclusion (RFI) is the ability to *include* files from external websites. This usually occurs due to improper input validation.

include - in this context, this usually means *execute*

RFI is extremely dangerous and usually allows arbitrary code execution, but it doesn’t appear often in real websites.

In PHP, `allow_url_include` is a configuration option controlling whether RFI-prone URLs will be parsed in calls to `include(..)`. It is usually set to 0, so RFI will not work via this function.

Local File Inclusion

Local File Inclusion (LFI) is when you can access or execute files present on the server that you aren't supposed to.

Especially in combination with other vulnerabilities (e.g. a race condition - more on this later), it can be used to gain code execution. LFI is generally more difficult to gain RCE from than RFI, because the malicious code needs to be accessible locally on the target server.

When you have LFI on a server, you usually look for files which contain valuable information: source code (/var/www/html/index.php), user accounts (/etc/passwd), configuration files (depends :), etc.

Race Conditions

Unintended behaviour of software caused by two or more components interacting in unexpected ways, usually very quickly - **racing**.

We call an operation **atomic** if it seems to the outside world to execute in a single step, i.e. its intermediate state is not observable.

Race conditions are caused by non-atomic operations whose intermediate state can be observed and exploited.

This is a general class of bugs existing in all software, but we will focus on how it manifests in web applications.

A Contrived Example

```
# assume 'db' is a database backend on the server

@app.route('/login', methods=['GET'])
def login(name, password):
    record = db.get_user(name)
    if record is None:
        abort(400)
    if record['password'] == password:
        session.authenticated = True # authenticate the user
    ...

def add_admin_user(name, password):
    db.add_user(name) # user record initialized with: .username = name
                      #                               .password = ""
    db.set_user_parameter(name, 'is_admin', True)
    db.set_user_parameter(name, 'password', password)
```

What happens if `login` and `add_admin_user` run at the same time?
Assuming we can force `add_admin_user` to be executed at a known time with a known name and an unknown password, can we get access to the admin account?

A Contrived Example

```
# assume 'db' is a database backend on the server

@app.route('/login', methods=['GET'])
def login(name, password):
    record = db.get_user(name)
    if record is None:
        abort(400)
    if record['password'] == password:
        session.authenticated = True # authenticate the user
    ...

def add_admin_user(name, password):
    db.add_user(name) # user record initialized with: .username = name
                      #                               .password = ""
    db.set_user_parameter(name, 'is_admin', True)
    db.set_user_parameter(name, 'password', password)
```

`add_admin_user` is not atomic - the state with an admin user added but no password set (yet) can be observed and exploited!
In reality, database accesses are usually **transactional**, i.e. atomic.

Exploiting Race Conditions

To exploit a race condition, we need to send data quickly - usually a script which repeatedly tries to force the race is needed and the attack only succeeds after several tries.

Multithreading (e.g. in Python, the `threading` module) is useful, because it allows us to send multiple requests at the same time.

Bypassing File Filters

Legitimate file upload forms sanitize the inputs by verifying they do not contain executable code. This can sometimes be bypassed in clever ways:

Apache .htaccess files

Polyglots

PHP pseudo-URLS and filters (not covered today)

Apache `.htaccess`

Per-directory configuration file - overrides Apache settings. Usually disabled, depending on the system configuration. If enabled, a lot of things can be done with `.htaccess` control.

System configuration to enable:
https://httpd.apache.org/docs/2.4/howto/htaccess.html

Example malicious `.htaccess` which makes the server execute files with the .totallysafe extension as PHP:

```
AddType application/x-httpd-php .totallysafe
```

File Format Polyglots

Imagine a service that allow users to upload profile images. Properly written, the service will validate that the uploaded file is an actual, valid image.

BUT, what if we can create a file which is both an image AND a configuration file? Files which are correct when interpreted as several formats are called **polyglots**.

Here is a `script.py.php.cpp`:

```
1   #define print(x) int main() { puts("Hello C!"); return 0; }
2   #define WTF <?php echo "Hello PHP!"; die(); ?>
3   print("Hello Python!")
```

Workshop

https://school.sigint.mx/ - 4 challenges for today under “Web 4”

https://httpd.apache.org/docs/2.4/howto/htaccess.html - Apache docs

https://www.owasp.org/index.php/Testing_for_Local_File_Inclusion - LFI

https://www.owasp.org/index.php/Testing_for_Remote_File_Inclusion - RFI

https://github.com/ctfs/write-ups-2014/tree/master/hitcon-ctf-2014/polyglot - polyglot writeup from HITCON CTF 2014