Reversing Apple's syslogd bug

January 22, 2016 · 7 min · 1490 words · fG!

Two days ago El Capitan 10.11.3 was released together with security updates for Yosemite and Mavericks. The bulletin available here describes nine security issues, most of them related to kernel or **IOKit** drivers. The last security issue is about a memory corruption issue on **syslog** that could lead to arbitratry code execution with root privileges. I was quite curious about this bug mostly because it involved **syslogd**, a logging daemon.

This post is about reversing the vulnerability and finding how it could be exploited. Unfortunately for us Apple is very terse on its security updates – for example they say nothing about if it is exploitable on default OS X installations or requires particular conditions. As we will see later on, this bug is not exploitable on default OS X installations.

While Apple makes available the source code for many components used in OS X, most of the time there is a significant delay so we need to use binary diffing to find out the differences between the vulnerable and updated binary. The usual tool for this purpose is BinDiff but there is also a free alternative called Diaphora made by *Joxean Koret*. Both tools require *IDA* and on this post we are going to use **Diaphora**. For this purpose we will need a copy of the vulnerable and patched binaries. The easiest way is to copy the **syslogd** binary (found at

/usr/sbin/syslogd) before the updates are installed (usually it's a good idea to have virtual machines snapshots for each version) and then after (or just extract the new binary from the update packages – El Capitan, Yosemite, Mavericks). This post will focus on **Yosemite binaries**.

Diaphora essentially works by generating a database and then comparing its contents. Comparing the 10.11.2 and 10.11.3 syslogd binaries gets us the following warning from **Diaphora**:

This means that both binaries are very similar so we should expect minimal changes between the two. Only one change is detected and its output is below.

```
n1 sub_100008776 proc near
 2          push    rbp
 3          mov     rbp, rsp
 4          push    r15
 5          push    r14
 6          push    rbx
 7          push    rax             ; char
 8          mov     r14, rdi
 9          lea     r15, dword_10001DAD0
n10         movsxd  rsi, dword ptr [r15+28h]
 11         test    rsi, rsi
 12         jz      short loc_100008799
 13loc_100008793:
 14         mov     rdi, [r15+20h]
n15         jmp     short loc_1000087A3
 16loc_100008799:
 17         mov     qword ptr [r15+20h], 0
 18         xor     edi, edi        ; void *
n19loc_1000087a3:
 20         add     rsi, 4          ; size_t
 21         call    _reallocf
 22         mov     [r15+20h], rax
 23         test    rax, rax
 24         jz      short loc_1000087CD
 25loc_1000087b5:
 26         mov     ecx, [r14+20h]
 27         movsxd  rdx, dword ptr [r15+28h]
 28         lea     esi, [rdx+1]
 29         mov     [r15+28h], esi
 30         mov     [rax+rdx*4], ecx
 31         mov     ebx, [r15+28h]
 32         jmp     short loc_1000087E5
 33loc_1000087cd:
 34         lea     rdi, aAdd_lockdown_s; "add
    _lockdown_session: realloc failed\n"
 35         xor     ebx, ebx
 36         xor     eax, eax
n37         call    sub_100006937
 38         mov     dword ptr [r15+28h], 0
 39loc_1000087e5:
 40         mov     [r15+2Ch], ebx
 41         add     rsp, 8
 42         pop     rbx
 43         pop     r14
 44         pop     r15
 45         pop     rbp
 46         retn
t47sub_100008776 endp
```

```
n1 sub_100008772 proc near
 2          push    rbp
 3          mov     rbp, rsp
 4          push    r15
 5          push    r14
 6          push    rbx
 7          push    rax             ; char
 8          mov     r14, rdi
 9          lea     r15, dword_10001DAD0
n10         movsxd  rax, dword ptr [r15+28h]
 11         test    rax, rax
 12         jz      short loc_100008795
 13loc_10000878f:
 14         mov     rdi, [r15+20h]
n15         jmp     short loc_10000879F
 16loc_100008795:
 17         mov     qword ptr [r15+20h], 0
 18         xor     edi, edi        ; void *
n19loc_10000879f:
 20         lea     rsi, ds:4[rax*4]; size_t
 21         call    _reallocf
 22         mov     [r15+20h], rax
 23         test    rax, rax
 24         jz      short loc_1000087CD
 25loc_1000087b5:
 26         mov     ecx, [r14+20h]
 27         movsxd  rdx, dword ptr [r15+28h]
 28         lea     esi, [rdx+1]
 29         mov     [r15+28h], esi
 30         mov     [rax+rdx*4], ecx
 31         mov     ebx, [r15+28h]
 32         jmp     short loc_1000087E5
 33loc_1000087cd:
 34         lea     rdi, aAdd_lockdown_s; "add_loc

 35         xor     ebx, ebx
 36         xor     eax, eax
n37         call    sub_100006933
 38         mov     dword ptr [r15+28h], 0
 39loc_1000087e5:
 40         mov     [r15+2Ch], ebx
 41         add     rsp, 8
 42         pop     rbx
 43         pop     r14
 44         pop     r15
 45         pop     rbp
 46         retn
t47sub_100008772 endp
```

Legends

Colors	Links
Added	(f)irst
Changed	change
Deleted	(n)ext change
	(t)op

The change is quite subtle. The original code could be something like:

```
reallocf(pointer, value + 4);
```

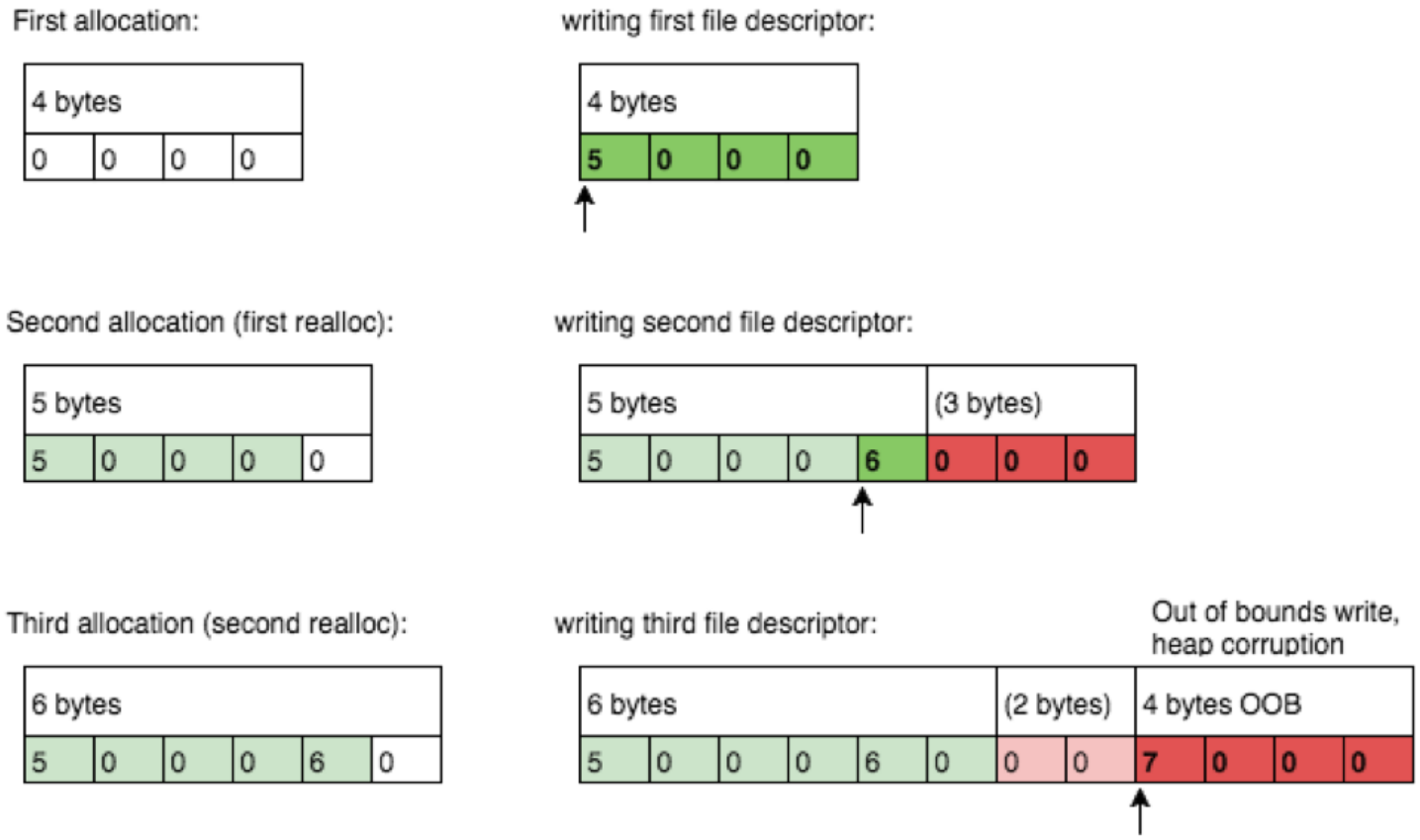
And the patch something like:

```
reallocf(pointer, value * 4 + 4);
```

The syslogd source package for El Capitan 10.11.2 can be downloaded here. The easiest way to try to locate this function is to grep the code using the string "**add_lockdown_session: realloc failed\n**", finding a single hit inside **syslogd.tproj/dbserver.c**. The source code for this function is:

```
void
add_lockdown_session(int fd)
{
  dispatch_once(&watch_init_once, ^{
    watch_queue = dispatch_queue_create("Direct Watch Queue", NULL);
  });

  dispatch_async(watch_queue, ^{
    if (global.lockdown_session_count == 0) global.lockdown_session_fds = NULL;

    global.lockdown_session_fds = reallocf(global.lockdown_session_fds, global.lockdown_session_count + 1 * sizeof(int));

    if (global.lockdown_session_fds == NULL)
    {
      asldebug("add_lockdown_session: realloc failed\n");
      global.lockdown_session_count = 0;
    }
    else
    {
      global.lockdown_session_fds[global.lockdown_session_count++] = fd;
    }

    global.watchers_active = direct_watch_count + global.lockdown_session_count;
  });
}
```

This makes it easier to observe the vulnerability. The patch is made on the **reallocf()** allocation size while the vulnerability is triggered when the **fd** variable is written into the **lockdown_session_fds** array. The allocation size used in **reallocf()** is wrong since **it's allocating memory just for the number of lockdown sessions instead of enough memory for each session**. The following image taken from Zimperium's analysis is a perfect illustration of the overflow and heap corruption.



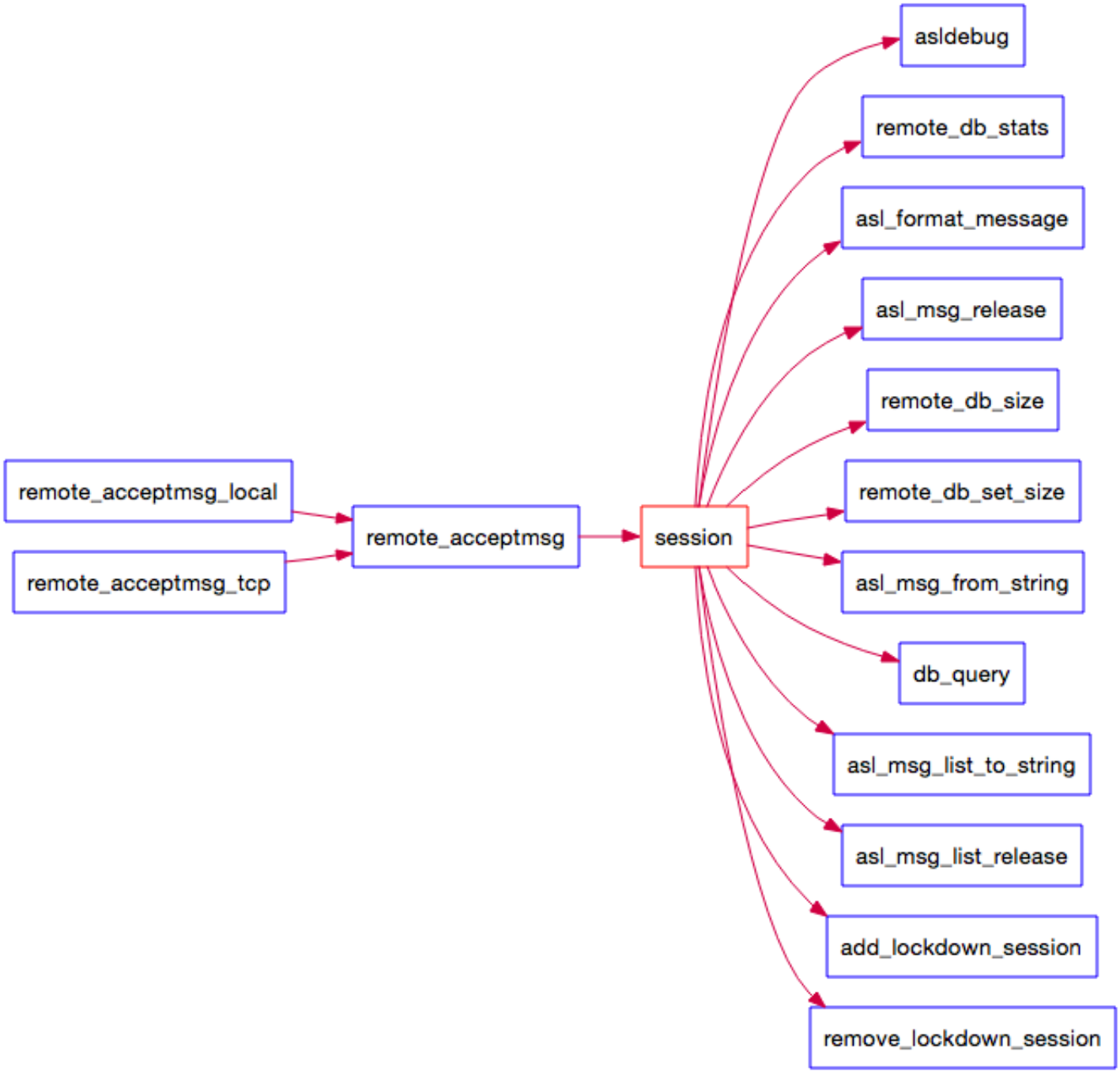
At the third connection the heap corruption is happening but from my tests more connections are required to make it crash (I get most of the time crashes in different areas than Zimperium but I was also testing against OS X).

The developer of this particular piece of code made a mistake, and the fix can be as simple as adding a set of parenthesis:

```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds, (global.lockdown_session_count + 1) * sizeof(int));
```

C language is powerful but unforgiving of these small mistakes.

At this point we know where the vulnerability is and how it was patched. The next question is how do we reach this function? The following is the partial call graph for **add_lockdown_session()**:



Judging by the initial function names the vulnerable function could be reached either locally (unix socket?) or remotely/locally (via TCP socket). The security bulletin mentions an attack from a local user. Looking at **/System/Library/LaunchDaemons/com.apple.syslogd.plist** configuration we can only observe the syslog unix socket:

```
$ plutil -p xml1 /System/Library/LaunchDaemons/com.apple.syslogd.plist
(...)
  "Sockets" => {
    "BSDSystemLogger" => {
      "SockPathName" => "/var/run/syslog"
      "SockType" => "dgram"
      "SockPathMode" => 438
    }
  }
(...)
```

This means that the **default configuration in OS X is not vulnerable**, unless the user changes it. Unfortunately for us Apple doesn't mention this in the bulletin, which is indeed interesting information for example to anyone running old systems that can't be upgraded. Let's dig a bit deeper and understand what do we need to do to activate this feature in OS X so we can try to reproduce the vulnerability. The **remote_acceptmsg_tcp()** function seems like a good candidate to trace back. Looking it up on source code we will find an interesting function:

```
int
remote_init(void)
{
  static dispatch_once_t once;

  dispatch_once(&once, ^{
    in_queue = dispatch_queue_create(MY_ID, NULL);
  });

  asldebug("%s: init\n", MY_ID);

#ifdef LOCKDOWN
  rfdl = remote_init_lockdown();
#endif

#ifdef REMOTE_IPV4
  rfd4 = remote_init_tcp(AF_INET);
#endif

#ifdef REMOTE_IPV6
  rfd6 = remote_init_tcp(AF_INET6);
#endif

  return 0;
}
```

This is the function that will activate the remote feature which allows us to reach the vulnerable code. The **#ifdef** means that we can check the binary to see if they were compiled or not into the final binary.

```
__text:0000000100000D2B3 remote_init     proc near               ; CODE XREF: start+8D2↓p
__text:0000000100000D2B3                                         ; DATA XREF: start+8A6↓o
__text:0000000100000D2B3                 push    rbp             ; char
__text:0000000100000D2B4                 mov     rbp, rsp
__text:0000000100000D2B7                 cmp     cs:qword_10001C9E0, 0FFFFFFFFFFFFFFFFh
__text:0000000100000D2BF                 jnz     short loc_10000D2EA
__text:0000000100000D2C1
__text:0000000100000D2C1 loc_10000D2C1:                          ; CODE XREF: remote_init+4A↓j
__text:0000000100000D2C1                 lea     rdi, aSInit     ; "%s: init\n"
__text:0000000100000D2C8                 lea     rsi, aRemote    ; "remote"
__text:0000000100000D2CF                 xor     eax, eax
__text:0000000100000D2D1                 call    asl_debug
__text:0000000100000D2D6                 mov     edi, 2
__text:0000000100000D2DB                 call    remote_init_tcp
__text:0000000100000D2E0                 mov     cs:dword_10001B7A8, eax
__text:0000000100000D2E6                 xor     eax, eax
__text:0000000100000D2E8                 pop     rbp
__text:0000000100000D2E9                 retn
__text:0000000100000D2EA ; ---------------------------------------------------------------------------
__text:0000000100000D2EA
__text:0000000100000D2EA loc_10000D2EA:                          ; CODE XREF: remote_init+C↑j
__text:0000000100000D2EA                 lea     rdi, qword_10001C9E0
__text:0000000100000D2F1                 lea     rsi, off_10001B250
__text:0000000100000D2F8                 call    _dispatch_once
__text:0000000100000D2FD                 jmp     short loc_10000D2C1
__text:0000000100000D2FD remote_init     endp
```

The disassembly output of **remote_init()** shows that only **remote_init_tcp()** was compiled, meaning that we can reach the vulnerable code via tcp sockets, either locally or remote depending on user configuration. The **remote_init_tcp()** function takes care of creating and binding the listener socket and is the one calling **remote_acceptmsg_tcp()** we saw in the first callgraph using **Grand Central Dispatch**.

```
int
remote_init_tcp(int family)
{
  (...)
  in_src_tcp = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ, (uintptr_t)fd, 0, in_queue);
  dispatch_source_set_event_handler(in_src_tcp, ^{ remote_acceptmsg_tcp(fd); });
  dispatch_resume(in_src_tcp);

  return fd;
}
```

We still don't know how to activate the remote feature. Next step is to see who calls **remote_init()**. There are two calls but the most interesting is **init_modules()**.

```
#if !TARGET_IPHONE_SIMULATOR
/* Interactive Module */
int remote_init(void);
int remote_reset(void);
int remote_close(void);
static int remote_enabled = 0;
#endif

static void
init_modules()
{
(...)
#if !TARGET_IPHONE_SIMULATOR
(...)
  /* remote (iOS support) module */
  m_remote = (module_t *)calloc(1, sizeof(module_t));
  if (m_remote == NULL)
  {
    asldebug("alloc failed (init_modules remote)\n");
    exit(1);
  }

  m_remote->name = "remote";
  m_remote->enabled = remote_enabled;
  m_remote->init = remote_init;
  m_remote->reset = remote_reset;
  m_remote->close = remote_close;

  if (m_remote->enabled) m_remote->init();
#endif /* TARGET_IPHONE_SIMULATOR */
(...)
}
```

The remote module support will be compiled into **syslogd** binary if the target is not the iOS simulator and the default enable or disable status depends on the **remote_enable** local variable. Its default value is zero, meaning **the remote feature is disabled by default**. This is another strong clue about a default OS X not being vulnerable.

Finally **init_modules()** is called by **main()**, where we can find the final clues about how to activate this feature.

```
int
main(int argc, const char *argv[])
{
(...)
#if TARGET_OS_EMBEDDED
  remote_enabled = 1;
  activate_bsd_out = 0;
#endif
(...)

  /* first pass sets up default configurations */
  for (i = 1; i < argc; i++)
  {
    if (streq(argv[i], "-config"))
    {
      if (((i + 1) < argc) && (argv[i+1][0] != '-'))
      {
        i++;
        if (streq(argv[i], "mac"))
        {
          global.dbtype = DB_TYPE_FILE;
          global.db_file_max = 25600000;
        }
        else if (streq(argv[i], "appletv"))
        {
          global.dbtype = DB_TYPE_FILE;
          global.db_file_max = 10240000;
        }
        else if (streq(argv[i], "iphone"))
        {
#if TARGET_IPHONE_SIMULATOR
          global.dbtype = DB_TYPE_FILE;
          global.db_file_max = 25600000;
#else
          global.dbtype = DB_TYPE_MEMORY;
          remote_enabled = 1;
#endif
        }
      }
    }
```

```
  }
(...)

#if !TARGET_IPHONE_SIMULATOR
    else if (streq(argv[i], "-bsd_out"))
    {
      if ((i + 1) < argc) activate_bsd_out = atoi(argv[++i]);
    }
    else if (streq(argv[i], "-remote"))
    {
      if ((i + 1) < argc) remote_enabled = atoi(argv[++i]);
    }
#endif
(...)

  asldebug("initializing modules\n");
  init_modules();

(...)
}
```

Inside main we can observe interesting things and finally be sure if OS X is vulnerable on default installation or not. The first thing we can observe in the above code snippet is that the remote feature is enabled by default on the embedded OS, usually meaning iOS and AppleTV. Next there is an option **-config** that also enables it if the **iphone** option is selected. Last is the undocumented **-remote** command line option, which can enable the remote feature on any Apple operating system.

To activate the feature we need to edit **syslogd** launchd configuration file found at **/System/Library/LaunchdDaemons/com.apple.syslogd.plist** (usually in binary format but can be converted using **plutil -convert xml1 filename**). The **ProgramArguments** and **Sockets** keys need to be modified to the following:

```
<key>ProgramArguments</key>
<array>
  <string>/usr/sbin/syslogd</string>
  <string>-remote</string>
  <string>1</string>
</array>
<key>Sockets</key>
<dict>
  <key>BSDSystemLogger</key>
  <dict>
    <key>SockPathMode</key>
    <integer>438</integer>
    <key>SockPathName</key>
    <string>/var/run/syslog</string>
    <key>SockType</key>
    <string>dgram</string>
  </dict>
  <key>Listeners</key>
  <dict>
    <key>SockNodeName</key>
    <string>syslog</string>
    <key>SockServiceName</key>
    <string>203</string>
    <key>SockType</key>
    <string>stream</string>
  </dict>
</dict>
```

Because **launchd** controls the sockets we also need to configure the socket where **syslogd** will be listening for the remote option (**#define ASL_REMOTE_PORT 203**). After we modify the plist and reload syslogd we can finally connect to port 203.

The vulnerable code path is triggered using the **watch** command. If we attach a debugger and insert a breakpoint in the vulnerable **add_lockdown_session()**, the breakpoint will never be hit when we select the **watch** command. This is the code inside session that calls the vulnerable function:

The **WATCH_LOCKDOWN_START** is only set in one place inside session:

The **SESSION_FLAGS_LOCKDOWN** is a flag passed on the only session argument.
And we can finally observe and conclude why the security bulletin talks about a local user:

This means that the **SESSION_FLAGS_LOCKDOWN** flag is only set on local connections and never on remote tcp connections, the only feature we have enabled in OS X **syslogd** binary. The functions who call **remote_acceptmsg()** show it clearly.

The conclusion is that **there is no code path to trigger this bug in OS X**, even if the user configures the remote feature. To test the bug and observe it in action the only way is to attach to the **syslogd** binary (or patch it) and remove the above condition (we can also patch inside session but here is easier). Next we just need a small tcp client that sends a few connections to the port 203 and issues the **watch** command. Sooner or later the **syslogd** binary will finally crash.
The vulnerability also doesn't seem easy to exploit because we don't have much control over the **fd** variable that is overwriting the allocated array.

One final note is that the vulnerability was only patched in El Capitan but not included in Yosemite and Mavericks security updates. While we have just seen that even on El Capitan there is no code path to the vulnerable code it is weird that the older versions weren't also patched. Apple security policy is still confusing most of the time.

So after a long post we can finally conclude that **there is nothing really interesting about this vulnerability in OS X** (and also iOS given the potential barriers to exploitation). It was just an interesting reverse engineering and source code analysis exercise to understand the vulnerability impact in OS X. This exercise wouldn't be needed if Apple just published more relevant details on its security bulletin.

Thanks to *pmsac* for draft post review and exploitation discussion (also to *qwertyoruiop* and *poupas*).

Have fun,
fG!

P.S.:
The tool used to generate the call graph is **Understand** from http://www.scitools.com. It's a great tool for browsing and auditing large projects.

vulnerability

« PREV
The Italian morons are back! What are they up to this time?

NEXT »
Gatekeerper – A kernel extension to mitigate Gatekeeper bypasses