Linux Based Inter-Process Code Injection Without Ptrace(2)

Tuesday, September 5, 2017 At 5:01AM

Using the default permission settings found in most major Linux distributions it is possible for a user to gain code injection in a process, without using ptrace. Since no syscalls are required using this method, it is possible to accomplish the code injection using a language as simple and ubiquitous as Bash. This allows execution of arbitrary native code, when only a standard Bash shell and coreutils are available. Using this technique, we will show that the noexec mount flag can be bypassed by crafting a payload which will execute a binary from memory.

The /proc filesystem on Linux offers introspection of the running of the Linux system. Each process has its own directory in the filesystem, which contains details about the process and its internals. Two pseudo files of note in this directory are maps and mem. The maps file contains a map of all the memory regions allocated to the binary and all of the included dynamic libraries. This information is now relatively sensitive as the offsets to each library location are randomised by ASLR. Secondly, the mem file provides a sparse mapping of the full memory space used by the process. Combined with the offsets obtained from the maps file, the mem file can be used to read from and write directly into the memory space of a process. If the offsets are wrong, or the file is read sequentially from the start, a read/write error will be returned, because this is the same as reading unallocated memory, which is inaccessible.

The read/write permissions on the files in these directories are determined by the ptrace_scope file in /proc/sys/kernel/yama, assuming no other restrictive access controls are in place (such as SELinux or AppArmor). The Linux kernel offers documentation for the different values this setting can be set to. For the purposes of this injection, there are two pairs of settings. The lower security settings, 0 and 1, allow either any process under the same uid, or just the parent process, to write to a processes /proc/${PID}/mem file, respectively. Either of these settings will allow for code injection. The more secure settings, 2 and 3, restrict writing to admin-only, or completely block access respectively. Most major operating systems were found to be configured with '1' by default, allowing only the parent of a process to write into its /proc/${PID}/mem file.

This code injection method utilises these files, and the fact that the stack of a process is stored inside a standard memory region. This can be seen by reading the maps file for a process:

```
$ grep stack /proc/self/maps
7ffd3574b000-7ffd3576c000 rw-p 00000000 00:00 0                          [stack]
```

the process is in an expected state when we are overwriting the stack, we use the sleep command as the slave process which is overwritten. The sleep command uses the nanosleep syscall internally, which means that the sleep command will sit inside the same function for almost its entire life (excluding setup and teardown). This gives us ample opportunity to overwrite the stack of the process before the syscall returns, at which point we will have taken control with our manufactured chain of ROP gadgets. To ensure that the location of the stack pointer at the time of the syscall execution, we prefix our payload with a NOP sled, which will allow the stack pointer to be at almost any valid location, which upon return will just increase the stack pointer until it gets to and executes our payload.

A general purpose implementation for code injection can be found at https://github.com/GDSSecurity/Cexigua. Efforts were made to limit the external dependencies of this script, as in some very restricted environments utility binaries may not be available. The current list of dependencies are:

- GNU grep (Must support -Fao –byte-offset)
- dd (required for reading/writing to an absolute offset into a file)
- Bash (for the math and other advanced scripting features)

The general flow of this script is as follows:

Launch a copy of sleep in the background and record its process id (PID). As mentioned above, the sleep command is an ideal candidate for injection as it only executes one function for its whole life, meaning we won't end up with unexpected state when overwriting the stack. We use this process to find out which libraries are loaded when the process is instantiated.

Using /proc/${PID}/maps we try to find all the gadgets we need. If we can't find a gadget in the automatically loaded libraries we will expand our search to system libraries in /usr/lib. If we then find the gadget in any other library we can load that library into our next slave using LD_PRELOAD. This will make the missing gadgets available to our payload. We also verify that the gadgets we find (using a naive 'grep') are within the .text section of the library. If they are not, there is a risk they will not be loaded in executable memory on execution, causing a crash when we try to return to the gadget. This 'preload' stage should result in a possibly empty list of libraries containing gadgets missing from the standard loaded libraries.

Once we have confirmed all gadgets can be available to us, we launch another sleep process, LD_PRELOADing the extra libraries if necessary. We now re-find the gadgets in the libraries, and we relocate them to the correct ASLR base, so we know their location in the memory space of the target region, rather than just the binary on disk. As above, we verify that the gadget is in an executable memory region before we commit to using it.

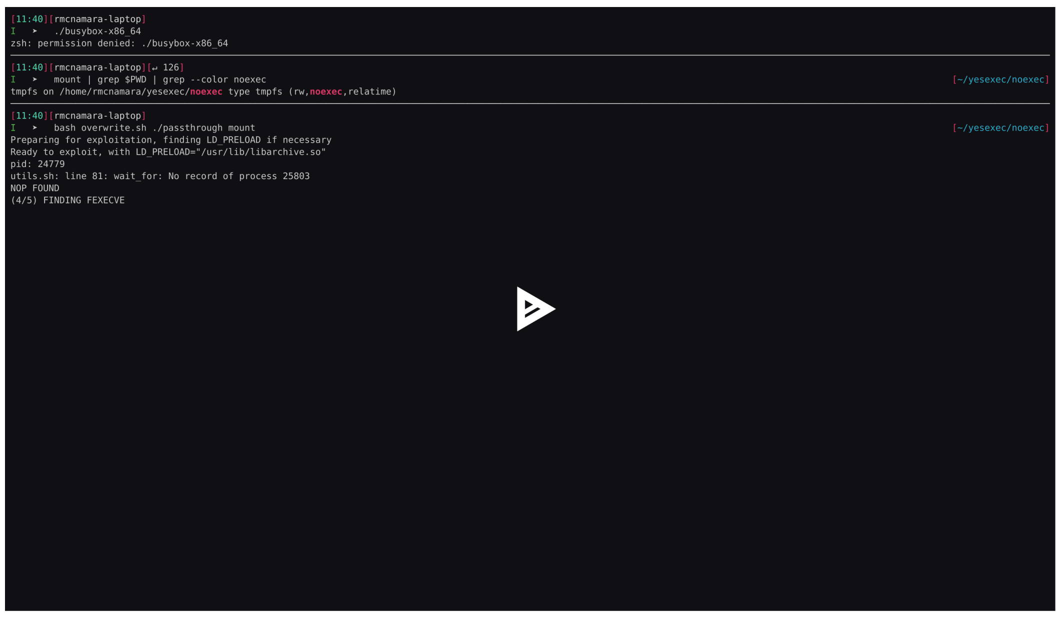
The list of gadgets we require is relatively short. We require a NOP for the above discussed NOP sled, enough POP gadgets to fill all registers required for a function call, a gadget for we can append strings to our payload and add pointers to them as necessary. For example, the fexecve syscall requires a char** for the arguments array. We can generate the array of pointers before injection inside our payload and upon execution the pointer on the stack to the array of pointers can be used as with a normal stack allocated char**.

Once the payload has been fully serialized, we can overwrite the stack inside the process using dd, and the offset to the stack obtained from the /proc/${PID}/maps file. To ensure that we do not encounter any permissions issues, it is necessary for the injection script to end with the 'exec dd' line, which replaces the bash process with the dd process, therefore transferring parental ownership over the sleep program from bash to dd.

After the stack has been overwritten, we can then wait for the nanosleep syscall used by the sleep binary to return, at which point our ROP chain gains control of the application and our payload will be executed.

The specific payload to be injected as a ROP chain can reasonably be anything that does not require runtime logic. The current payload in use is a simple open/memfd_create/sendfile/fexecve program. This disassociates the target binary with the filesystem noexec mount flag, and the binary is then executed from memory, bypassing the noexec restriction. Since the sleep binary is backgrounded on execution by bash, it is not possible to interact with the binary to be executed, as it does not have a parent after dd exits. To bypass this restriction, it is possible to use one of the examples present in the libfuse distribution, assuming fuse is present on the target system: the passthrough binary will create a mirrored mount of the root filesystem to the destination directory. This new mount is not mounted noexec, and therefore it is possible to browse through this new mount to a binary, which will then be executable.

A proof of concept video shows this passthrough payload allowing execution of a binary in the current directory, as a standard child of the shell.



```
[11:40][rmcnamara-laptop]
I  ➤  ./busybox-x86_64
zsh: permission denied: ./busybox-x86_64

[11:40][rmcnamara-laptop][↵ 126]
I  ➤  mount | grep $PWD | grep --color noexec                                   [~/yesexec/noexec]
tmpfs on /home/rmcnamara/yesexec/noexec type tmpfs (rw,noexec,relatime)

[11:40][rmcnamara-laptop]
I  ➤  bash overwrite.sh ./passthrough mount                                      [~/yesexec/noexec]
Preparing for exploitation, finding LD_PRELOAD if necessary
Ready to exploit, with LD_PRELOAD="/usr/lib/libarchive.so"
pid: 24779
utils.sh: line 81: wait_for: No record of process 25803
NOP FOUND
(4/5) FINDING FEXECVE
```

Future work:

normal user is not in use. This value can be set by adding the following line to /etc/sysctl.conf:

```
kernel.yama.ptrace_scope=2
```

Other mitigation strategies include combinations of Seccomp, SELinux or Apparmor to restrict the permissions on sensitive files such as /proc/${PID}/maps or /proc/${PID}/mem.

The proof of concept code, and Bash ROP generator can be found at https://github.com/GDSSecurity/Cexigua

Author: Rory McNamara
©Aon plc 2023

Gotham Digital Science, LLC (GDS), is a wholly owned subsidiary of Aon, PLC

About Cyber Solutions
Cyber security services are offered by Stroz Friedberg Inc., its subsidiaries and affiliates. Stroz Friedberg is part of Aon's Cyber Solutions which offers holistic cyber risk management, unsurpassed investigative skills, and proprietary technologies to help clients uncover and quantify cyber risks, protect critical assets, and recover from cyber incidents.