Yasanka Sameera Horawalavithana

Smash Stack

This type of exploits caused due to the usual programming error which allocates sufficiently large data to an array without checking for index boundaries. This leads to crash the user program, overwrite the stack registers, and enable the program to execute any arbitrary instruction at an arbitrary location. An adversary could hijack this flow of instructions to do whatever he wants, usually exploiting to an interactive shell with root privileges.

Think like an attacker

Buffer overflow could be thought as an trigger to execute any arbitrary flow of control that an adversary wishes to proceed with. In this exploit, the ultimate goal is to get into a shell with root privileges. What is the approach of an adversary? An adversary wants to construct a “sufficiently large” buffer with instruction of the flow, which would be triggered by the execution of vulnerable program. In our case, getscore is the program that we exploit vulnerability.

First, let’s understand how the vulnerability looks like in getscore program. The overflow of buffer is caused by the copy of arbitrary string values to the array of matching_pattern, that has only 128 bytes allocated. get_score subroutine has two other pointers allocated 4 bytes each. Any arbitrary string that exceeds the given memory allocation could overwrite the base pointer (EBP) of the stack frame and return address of the subroutine (EIP). The goal is to overwrite the return address with the correct location of the place where our shell code lives in the memory.

Second, we want to place our shell code in the memory to be executed once we exploit the return address of the subroutine. Here, we develop two versions of exploit;

- **Redhat 8: with direct jump to shell code (Figure 1)**

```
./exploit_gen_with_esp <ESP Value> <EGG Size> <Offset>
```

In this version, given the ESP and offset, we set the return address (EIP) to a location where it’s possible to execute shell code. How to know the exact sizes of buffer? The size of shell code is fixed, so we need to have a buffer with minimum size that could contain the shell code and return addresses. We know the buffer we’re going to overflow in getscore program is 128 bytes. With other pointer and register values, we need to have a buffer of size at least 144 bytes. Here, we construct a buffer of size 160 bytes, place NOP instructions in the first half of the buffer, then followed by the shell code, making enough space for the return addresses. Ideally, we expect the EIP value to be overwritten with the return addresses which would point to NOP instruction, and execute shell code eventually.

We face several difficulties to get to know the exact ESP address. We would get different ESP addresses in multiple runs depending on the environment variable. The current approach is to get a segmentation fault with dummy data as input, analyze the core files, and find the ESP address to construct the buffer that we use for the exploit. This version works well in Redhat8, but not in RedHat9 due to stack address randomization.

- **RedHat 8 and 9: with indirect jump to mitigate stack address randomization (Figure 2)**

```
./exploit_gen_with_jmp_esp <ldd entry address> <EGG Size>
```

We use a similar approach in this version with the previous one, but this version is more reliable because it doesn’t need any ESP value as an explicit input. We search address space starting with ldd entry address to find the location for `JMP ESP` address, which is used as our return address to point to the location of shell code. Importantly, we place our shell code starting at ESP address. How to know the exact sizes of buffer? Since, we place our shell code after ESP, we know we have to overflow the buffer by the size of shell code at least. So, we use the minimum buffer size as 190 bytes.

Note: We attached the example output files in the submission, as well as figures explaining our exploit program at Figure 3 and 4.

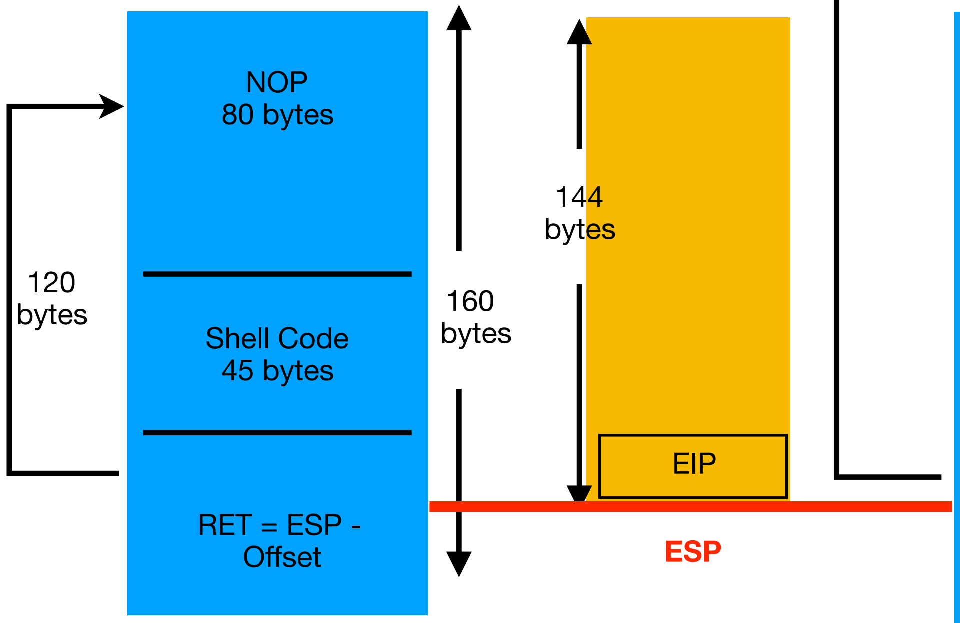


Figure 1

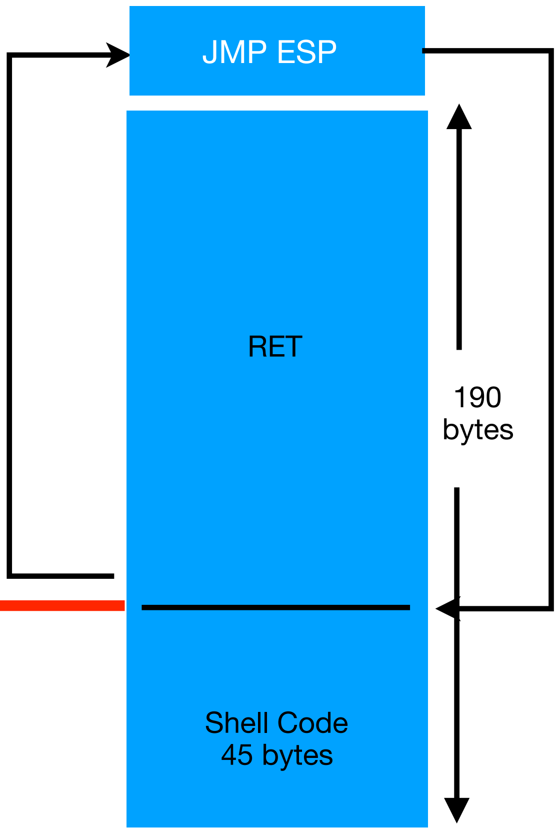
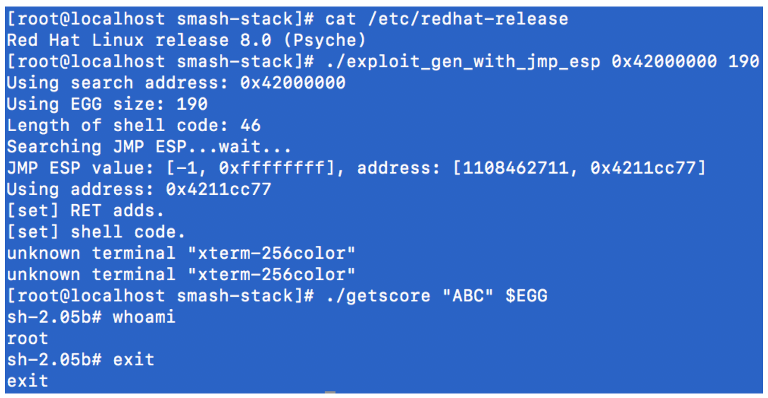
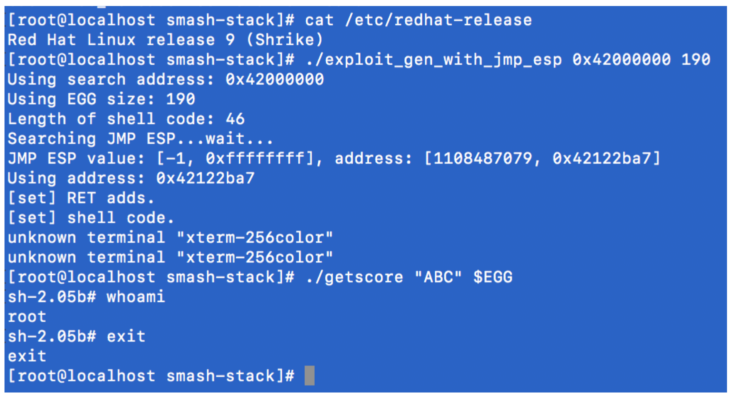


Figure 2



```
[root@localhost smash-stack]# cat /etc/redhat-release
Red Hat Linux release 8.0 (Psyche)
[root@localhost smash-stack]# ./exploit_gen_with_jmp_esp 0x42000000 190
Using search address: 0x42000000
Using EGG size: 190
Length of shell code: 46
Searching JMP ESP...wait...
JMP ESP value: [-1, 0xffffffff], address: [1108462711, 0x4211cc77]
Using address: 0x4211cc77
[set] RET adds.
[set] shell code.
unknown terminal "xterm-256color"
unknown terminal "xterm-256color"
[root@localhost smash-stack]# ./getscore "ABC" $EGG
sh-2.05b# whoami
root
sh-2.05b# exit
exit
```

Figure 3: exploit_gen_with_jmp_esp; Redhat 8



```
[root@localhost smash-stack]# cat /etc/redhat-release
Red Hat Linux release 9 (Shrike)
[root@localhost smash-stack]# ./exploit_gen_with_jmp_esp 0x42000000 190
Using search address: 0x42000000
Using EGG size: 190
Length of shell code: 46
Searching JMP ESP...wait...
JMP ESP value: [-1, 0xffffffff], address: [1108487079, 0x42122ba7]
Using address: 0x42122ba7
[set] RET adds.
[set] shell code.
unknown terminal "xterm-256color"
unknown terminal "xterm-256color"
[root@localhost smash-stack]# ./getscore "ABC" $EGG
sh-2.05b# whoami
root
sh-2.05b# exit
exit
[root@localhost smash-stack]#
```

Figure 4: exploit_gen_with_jmp_esp; Redhat 9