Buffer Overflow Techniques and Exploits

Karan Agarwalla	:	180050045
Mohammad Ali Rehan	:	180050061
Neel Aryan Gupta	:	180050067
Shreya Pathak	:	180050100

Introduction

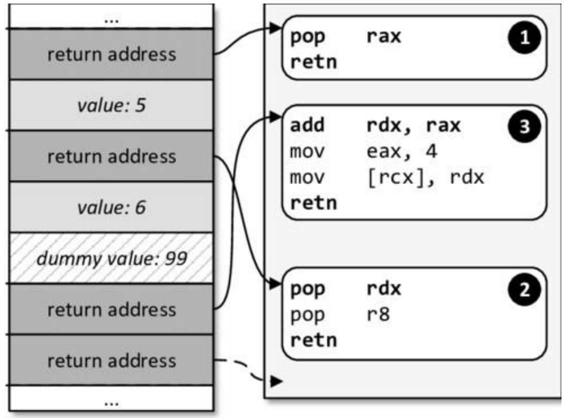
The execution stack is an indispensable component of modern operating systems and architectures. We have studied how the lack of certain securities and use of unchecked buffers can be successfully used to take control of any machine or server and possibly gain further root access.

We present 3 attacks which all belong to Return Oriented Programming (ROP), each designed with one goal in mind, to spawn a (*sh*) shell at the target machine (or server). These attacks aim to redirect code execution by exploiting the *ret* instruction.

We also present case studies of how this technique has been used in the past for mass scale attacks leading to DDoS and infiltration of user systems across the globe.

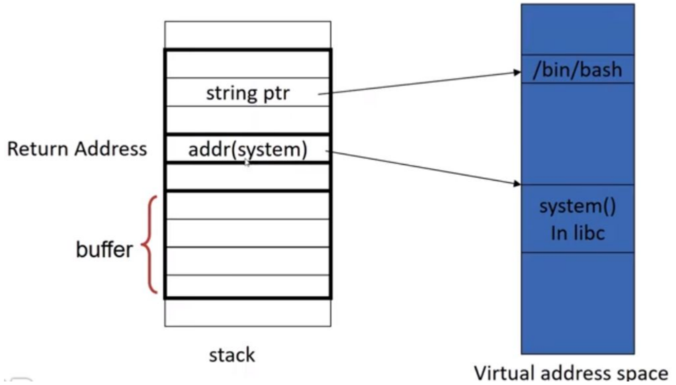
ret2syscall

We make use of the fact that executable is compiled statically and hence the assembly code of the executable has a lot of usable gadgets. We use a tool like *ROPgadget* to find gadgets pre-existing in the binary. Gadgets help us to execute simple instructions like *move*, *pop* and then a *ret* instruction. We design a chain of address of gadgets, which can be executed one after the other and perform a *syscall* in the end.



ret2libc

In ret2libc, the executable has been dynamically linked to libc, hence has fewer gadgets preventing us from executing a ret2syscall attack. Firstly, the GOT table entry value for a known function is leaked using *puts()* to help us calculate libc's base address. The return address placed is that of *puts()* for which the return address is *vuln()* again allowing us to overflow the buffer in the *read()* function call again. With this libc base address, call the *system('/bin/sh')* using offsets of functions from libc.



ret2dlresolve

In ret2dlresolve attack, the executable is dynamically linked to libc and there are no *puts()* function calls in the user code preventing us from executing ret2syscall and ret2libc attacks respectively. Through this technique, we exploit the lazy binding mechanism, which is done at runtime, and replace the GOT entry value of *read()* with that of *system()* function.

To do this we first forge the different structs involved in fetching symbol information by the runtime resolver *_dl_runtime_resolve* and place them in BSS segment. We also place the strings *system* and */bin/sh* in the BSS segment. Then we call the *runtime_resolver* with appropriately forged arguments to replace the GOT entry of a function (here *read()*). The call to resolver automatically executes our replaced function (here *system()* whose argument */bin/sh* was put on the stack as a part of the payload) as it thinks this function/symbol has been called for the first time in the code.

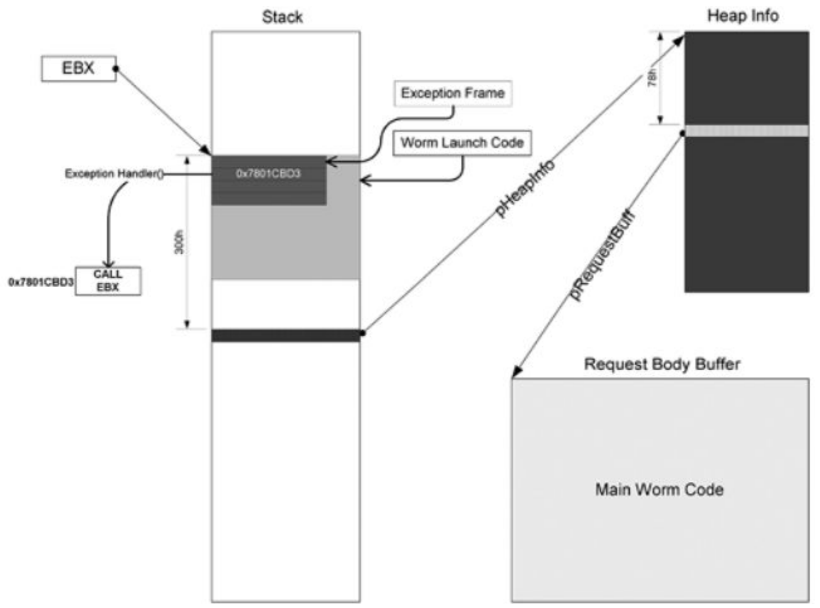
Morris Worm

Morris Worm was launched in the 80s and employed several techniques to leak sensitive data. One technique targeted the finger daemon which was run on many servers for accessing client information and had an unchecked buffer. This was the key to the attack. The worm overflowed the 512B buffer and overwrote the return address on the stack to execute malicious code placed on the stack by the worm. Below is the content it placed in the buffer to make the system call:

```
VAX Opcode          Assembly                    Comment
DD8F2F736800        pushl       $68732f         ; '/sh\0'
DD8F2F62696E        pushl       $6e69622f       ; '/bin'
D05E5A              movl        sp, r10         ; save pointer to command
DD00                pushl       $0              ; third parameter
DD00                pushl       $0              ; second parameter
DD5A                pushl       r10             ; push address of '/bin/sh\0'
DD03                pushl       $3              ; number of arguments for chmk
D05E5C              movl        sp, ap          ; Argument Pointer register
                                                ; = stack pointer
BC3B                chmk        $3b             ; change-mode-to-kernel
```

CodeRed Worm

The Worm exploited unchecked buffers present in the dynamic library *idq.dll*. This was used to receive HTTP GET requests from users and allowed users to search for documents by keyword. The worm sent a cleverly crafted string as URL to overwrite the address in the exception frame on the stack and make the EIP eventually point to the stack. This URL also doubled up as instructions for launching other actions like spreading further, or installing trojans(in CodeRed II).



```
0:  90                    nop
1:  90                    nop
2:  58                    pop    eax
3:  68 d3 cb 01 78        push   0x7801cbd3
8:  90                    nop
9:  90                    nop
a:  58                    pop    eax
b:  68 d3 cb 01 78        push   0x7801cbd3
10: 90                    nop                       <- ebx
(initially)
11: 90                    nop
12: 58                    pop    eax
13: 68 d3 cb 01 78        push   0x7801cbd3
18: 90                    nop
19: 90                    nop
1a: 90                    nop
1b: 90                    nop
1c: 90                    nop
1d: 81 c3 00 03 00 00     add    ebx, 0x300
23: 8b 1b                 mov    ebx, DWORD PTR [ebx]
25: 53                    push   ebx
26: ff 53 78              call   DWORD PTR [ebx+0x78]
29: 00 00                 add    BYTE PTR [eax],al
```

Challenges Faced & Takeaways

- The binary securities had to be carefully assessed for each type of attack, to see their effect on the overflow and the mechanism of the attack. Exactly those securities were kept ON which the particular attack technique can bypass.
- The assembly source code for CodeRed wasn't freely available so we had to recreate it from the URL using tools to get x86 instruction from the machine code.
- The BSD Unix 6 which the morris worm attacked, has been patched up. So it was not straightforward to figure what was the exact functionality that was being attacked, however some archives and reading the source code of relevant portion helped us find it.

❖ Through this project, we gained a deeper understanding of how the execution stack can be particularly dangerous when enough protections are not used

❖ As programmers, it teaches how one must be always careful in designing interactive programs and libraries. Even a single flaw is enough to infiltrate a machine

❖ We learnt about the many security provisions that are in place like NX bit, canary etc. to prevent successful execution of malicious code

❖ As a side, we also got to explore other related topics like DDoS or dynamic linking.