#4

PB19111701

1

Question:

Describe what a NOP sled is and how it is used in a buffer overflow attack.

Answer:

A NOP sled is a trick for writing robust exploits by adding a sequence of NOPs before the actual shellcode. With the help of a NOP sled, landing anywhere in the sled will eventually bring the execution of the program to the shellcode after several NOP instructions which do nothing at all. Consequently, in a buffer overflow attack, RIP of the previous stack frame needs only being set as one of the addresses in the NOP sled. That is to say, the attack only needs to be aware of the approximate address of the beginning of the shellcode instead of the exact one for successfully carrying out a buffer overflow attack.

2

Question:

Look into different shellcodes released in Packet Storm, and summarize different operations an attacker may design shellcode to perform.

Answer:

1. to break access control, e.g.

   Windows/x86 Add User Alfred Shellcode

   NTLM BITS SYSTEM Token Impersonation

   Linux/x86 Add Root User Shellcode

   Solaris/SPARC chmod() Shellcode

2. to escape security protections (such as firewall, sandbox), e.g.

   Microsoft Windows Firewall Disabling Shellcode

   Safari Type Confusion / Sandbox Escape

3. to execute an unauthenticated code or to download an unauthenticated file, e.g.

   Solaris/SPARC execve() Shellcode

   Windows/x86 Download File / Execute Shellcode

   Cisco RV340 SSL VPN Unauthenticated Remote Code Execution

4. egg hunter, i.e. a small shellcode for searching and executing another big shellcode when the exploitable buffer is too small for the big one, e.g.

   Linux/x86 Egghunter Shellcode

   Linux/x64_86 Egghunter Execve Shellcode

5. to change network configurations, e.g.

   [Windows/x86 Bind TCP Shellcode]

   [Windows/x86 Reverse TCP Shellcode]

   [Linux/ARM TCP Port 4444 Binding Shellcode]

6. to encrypt files, e.g.

   [Linux/x86 OpenSSL File Encryption Shellcode]

7. to crash the operating system, e.g.

   [Linux/x64 Kill All Processes Shellcode]

etc.

3

Question:

Below is a simple C code with a buffer overflow issue.

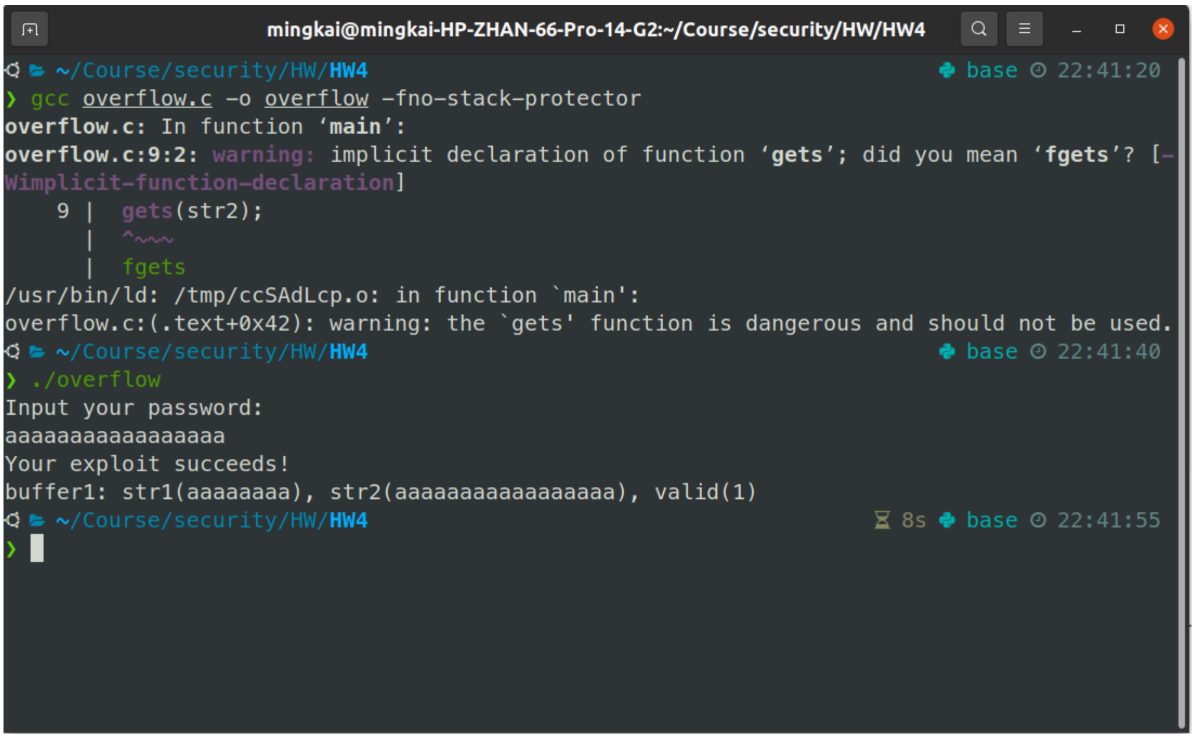
```c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
    int valid = false;
    char str1[9] = "fdalfakl";
    char str2[9];
    printf("Input your password:\n");
    gets(str2);
    if (strncmp(str1, str2, 8) == 0) {
        valid = true;
        printf("Your exploit succeeds!\n");
    }
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

a

Question:

Craft a simple buffer overflow exploit, and circumvent the password checking logic. Include in your submission necessary step-by-step screenshots or descriptions to demonstrate how you carry out the attack.

Answer:


mingkai@mingkai-HP-ZHAN-66-Pro-14-G2:~/Course/security/HW/HW4

```
~/Course/security/HW/HW4                                         base  22:41:20
❯ gcc overflow.c -o overflow -fno-stack-protector
overflow.c: In function ‘main’:
overflow.c:9:2: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
    9 |  gets(str2);
      |  ^~~~
      |  fgets
/usr/bin/ld: /tmp/ccSAdLcp.o: in function `main':
overflow.c:(.text+0x42): warning: the `gets' function is dangerous and should not be used.
~/Course/security/HW/HW4                                         base  22:41:40
❯ ./overflow
Input your password:
aaaaaaaaaaaaaaaaa
Your exploit succeeds!
buffer1: str1(aaaaaaaa), str2(aaaaaaaaaaaaaaaaa), valid(1)
~/Course/security/HW/HW4                                    8s   base  22:41:55
❯
```

b

Question:

Describe how to fix this buffer overflow issue.

Answer:

Use `fgets` instead of `gets`, so that the library will check the input and discard the remaining part which is out of the size of the array.

4

Question:

Alice is attacking a buggy application. She has found a vulnerability that allows her to control the values of the registers ecx, edx, and eip, and also allows her to control the contents of memory locations 0x9000 to 0x9014. She wants to use return-oriented programming, but discovers that the application was compiled without any ret instructions! Nonetheless, by analyzing the application, she learns that the application has the following code fragments (gadgets) in memory:

```
0x3000: add edx, 4       ; edx = edx + 4
        jmp [edx]        ; jump to *edx

0x4000: add edx, 4       ; edx = edx + 4
        mov eax, [edx]   ; eax = *edx
        jmp ecx          ; jump to ecx

0x5000: mov ebx, eax     ; ebx = eax
        jmp ecx          ; jump to ecx

0x6000: mov [eax], ebx   ; *eax = ebx
        ...              ; don't worry about what happens after this
```

Show how Elizabeth can set the values of the registers and memory so that the vulnerable application writes the value 0x2222 to memory address 0x8888.

Answer:

ecx	0x3000
edx	0x9000
eip	0x4000

0x9000	
0x9004	0x2222
0x9008	0x5000
0x900c	0x4000
0x9010	0x8888
0x9014	0x6000

5

Question:

Consider the following simplified code that was used earlier this year in a widely deployed router. If hdr->ndata = "ab" and hdr->vdata = "cd" then this code is intended to write "ab:cd" into buf. Suppose that the attacker has full control of the contents of hdr. Explain how this code can lead to an overflow of the local buffer buf.

```
uint32_t nlen, vlen;    /* values in 0 to 2^32-1 */
char buf[8264];

nlen = 8192;
if ( hdr->nlen <= 8192 )
    nlen = hdr->nlen;

memcpy(buf, hdr->ndata, nlen);
buf[nlen] = ':';

vlen = hdr->vlen;
if (8192 - (nlen+1) <= vlen )   /* DANGER */
    vlen = 8192 - (nlen+1);

memcpy(&buf[nlen+1], hdr->vdata, vlen);
buf[nlen + vlen + 1] = 0;
```

Answer:

Set `hdf->nlen` as a `unit32_t` s.t. its value is bigger than or equal to `8192`. In this case, `nlen` will be set as `8192` before the second `if` statement. Since the type of `nlen` is `unit32_t`, i.e. the value of `nlen` will be treated as an unsigned integer, `8192 - (nlen+1)` is regared as $2^{32}-1$ (that is -1 if regared as a signed integer). Consequently, regardless of the value of `vlen`, the condiction expression of the second `if` statement will always be false. Thus the value of `vlen` is bound to be set as the same as the value of `hdr->vlen`. Just set `hdf->vlen` as a `unit32_t` s.t. its value is bigger than or equal to `72`. There would be a buffer overflow after the execution of the program.