Software vulnerabilities

ROP

lab manual

Julien Bachmann

5 June 2015

Objectives

Objectives for this practical work are multiple: the student should get some familiarities with debugger and operating system internals, as well as learn security-oriented programming tricks.

Furthermore, he will have the opportunity to analyse the security of a custom binary and exploit the found vulnerability using an advanced technique that has become a standard nowadays.

Environment installation

For this practical work, we will use the Kali Linux distribution which is based on Ubuntu Linux. In order to safeguard the students work, the virtual machine version of Kali will be used instead of a live-cd.

If not already done, the virtual machine can be downloaded under the following link:

https://www.offensive-security.com/kali-linux-vmware-arm-image-download/

Please download the version corresponding to the hypervisor you are using. Being either VMware or VirtualBox.

1. Once booted, log-in the VM using the following credentials:
   root / toor

2. Check the VM Internet connectivity using the following command

```
# ping 8.8.8.8
```

3. Install PEDA extension for GDB

```
# cd ~ && mkdir local && cd local
# git clone https://github.com/longld/peda.git
# echo source ~/local/peda/peda.py >> ~/.gdbinit
```

4. Install *radare2* and its GTK interface, *bokken*

```
# apt-get install radare2 python-radare2 bokken
```

5. Install the compiled version of rp++

```
# cd ~/local/ && wget https://github.com/downloads/
0vercl0k/rp/rp-lin-x64
```

6. Install ROPGadget by cloning its Github repository

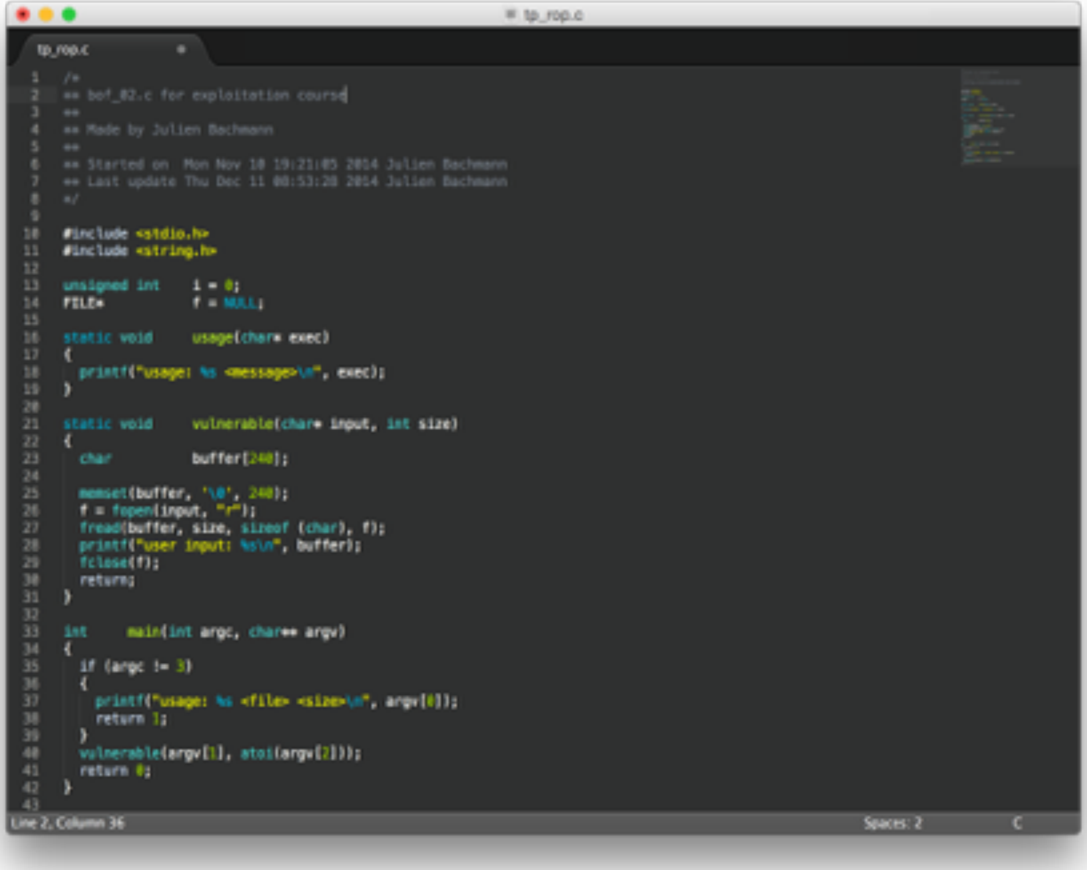
```
# cd ~/local/ && git clone https://github.com/
JonathanSalwan/ROPgadget.git
```

Program analysis

During this practical work the program named *tp_rop* will be analysed and probably exploited :) Its corresponding source code can be found in the *tp_rop.c* file.

Files for this lab can be retrieved at:

`https://github.com/milkmix-/training/tree/master/rop`



```c
/*
** bof_02.c for exploitation course
**
** Made by Julien Bachmann
**
** Started on  Mon Nov 10 19:21:05 2014 Julien Bachmann
** Last update Thu Dec 11 08:53:28 2014 Julien Bachmann
*/

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

unsigned int    i = 0;
FILE*           f = NULL;

static void     usage(char* exec)
{
  printf("usage: %s <message>\n", exec);
}

static void     vulnerable(char* input, int size)
{
  char          buffer[240];

  memset(buffer, '\0', 240);
  f = fopen(input, "r");
  fread(buffer, size, sizeof (char), f);
  printf("user input: %s\n", buffer);
  fclose(f);
  return;
}

int     main(int argc, char** argv)
{
  if (argc != 3)
  {
    printf("usage: %s <file> <size>\n", argv[0]);
    return 1;
  }
  vulnerable(argv[1], atoi(argv[2]));
  return 0;
}
```

The binary was compiled with the following command line:

```
$ gcc -fno-stack-protector -m32 -static tp_rop.c -o
tp_rop
```

1. Run the binary and try to understand its behaviour and describe it.

2. Open the binary with *radare2*. Analyse the binary (`aa` command). Disassemble the *sym.vulnerable* function (`pdf` command). Describe this function using the assembly code you retrieved.

```
[0x08048d0a]> aa
[0x08048d0a]> pdf @ sym.vulnerable
```

3. Now, open the source code file and match your understanding against the one you had using the assembly code.

4. Using the *man* page of the *read(2)* function, explain what this does, especially based on its arguments.

5. Based on the code, what does the user control from the outside of the application?

6. Describe the vulnerability in this program and how you plan to exploit it.

Vulnerability exploitation

1. Use `pattern_create.rb` to generate a long enough pattern and store it into a file.

2. Enable the creation of a core dump if not already done on your environment:

```
$ ulimit -c unlimited
```

3. Use the generated file as an input for `tp_rop` binary. Retrieve the address of the faulty instruction using `gdb` on the `core` file.

```
Address of the segmentation fault:
```

4. Use pattern_offset.rb to retrieve the offset at which the overflow occurs.

```
Offset for the overflow:
```

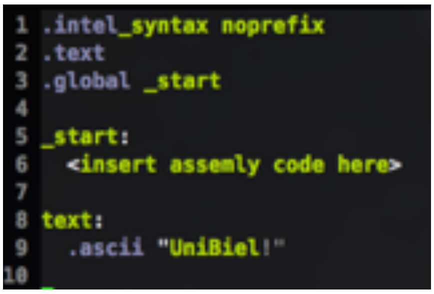
5. Use `radare2` to disassemble *sym.vulnerable* function. What is the assembly instruction that allows to retrieve the same result?

(screenshot or copy paste the instructions)

Before writing a full exploit allowing to get a shell, we will start by writing a ROP chain that write *UniBiel!* on `stdout`. As viewed during the course, lets start by writing the functionality in assembly.

6. What is the number of the `write` syscall?

7. Use write.s as a base to write a program that outputs *UniBiel!* on stdout.



```
.intel_syntax noprefix
.text
.global _start

_start:
  <insert assemly code here>

text:
  .ascii "UniBiel!"

```

```
$ as write.s -o write.o
$ ld write.o -o write
```

8. In the previous code you had your string directly in the binary. When using the ROP chain you will need to store and access this string at a fixed address in memory. In this case we will use the `.data` section which has `rw` rights.

Retrieve `.data` section address using `objdump`:

9. What gadgets can you use to store the 8 bytes (`UniBiel!`) in two times?

```
# ~/local/rp-lin-x64 -f ~/TPs/tp_rop -r 4
…
```

10. How can you set the syscall number previously found into `eax` using gadgets?

```
# ~/local/rp-lin-x64 -f ~/TPs/tp_rop -r 4
…
```

11. Now that you have the assembly and basic gadgets, create the ROP chain using gadgets found with rp++

```
# ~/local/rp-lin-x64 -f ~/TPs/tp_rop -r 4
…
```

12. Final step : now that you have written your ROP chain by hand, use `ROPGadget —ropchain` command line option to retrieve the one that allows to execute `/bin/sh`.