CON 2018

# Bug classification using r2

## beebug: A tool for checking exploitability

Andrea Sindoni

@invictus1306

# Who Am I ?



- ❑ Senior security researcher
- ❑ Specialized in reverse engineering and exploit development
- ❑ Have been working professionally in vulnerability research for more than 8 years
- ❑ I have never found any web vulnerability ☺

**Twitter**: @invictus1306

**Disclaimer**: All the work presented here is mine (not of my employer)

# Summary



- Motivation
- Common vulnerabilities
- Exploitable crashes
- Tool and usage
- Graph generation
- Demo
- Future direction
- Thank you

# Motivation

# Motivation

- If you work as vulnerability researcher, often you have to analyze a crash in order to understand:
  - what type of vulnerability you have found
  - whether it is exploitable
- How to **quickly** determine if a crash in a target is potentially exploitable
- Existing similar tools:
  - *exploitable.py* for gdb
  - *!exploitable* for WinDbg
- However, the tool presented here is easy to integrate into a automated script

# Common Vulnerabilities

# Common Vulnerabilities



- Heap Overflow (e.g., CVE 2018-11625)
- UAF (e.g., CVE-2018-11624)
- Stack Overflow (e.g., CVE-2018-12327)
- Integer Overflow (e.g., CVE-2017-15587)
- Type confusion (e.g., CVE-2018-4944)
- Out-of-bounds (e.g., CVE-2018-6149)

# Common Vulnerabilities

- Stack Overflow - **CVE-2018-12327** (ntpq 4.2.8p11 Local BufferOverflow)

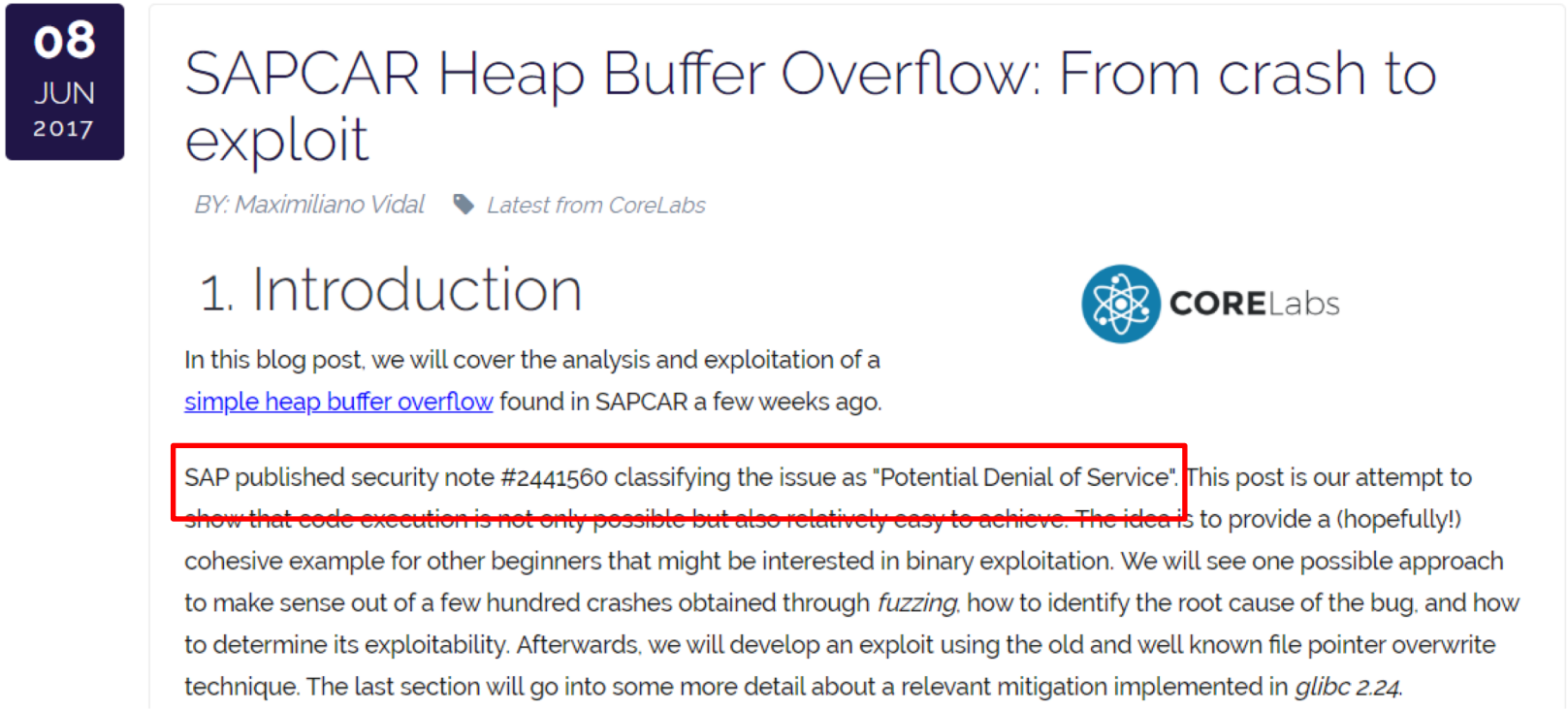
```
root@invictus1306-VirtualBox:/home/invictus1306/Downloads/ntp-4.2.8p11# ./ntpdc/ntpdc -4 [`python -c 'print "A" * 300'`]
Name or service not known
*** stack smashing detected ***: ./ntpdc/ntpdc terminated
Aborted (core dumped)
```

```
#define LENHOSTNAME     256             /* host name is 256 characters long */
```

```
static  int
openhost(
        const char *hname,
        int         fam
        )
{
        const char svc[] = "ntp";
        char temphost[LENHOSTNAME];
        int a_info, i;
        struct addrinfo hints, *ai;
        sockaddr_u addr;
        size_t octets;
        register const char *cp;
        char name[LENHOSTNAME];

        /*
         * We need to get by the [] if they were entered
         */

        cp = hname;

        if (*cp == '[') {
                cp++;
                for (i = 0; *cp && *cp != ']'; cp++, i++)
                        name[i] = *cp;
                if (*cp == ']') {
                        name[i] = '\0';
```

# Common Vulnerabilities

- Some crashes are not easy to classify and many times these are classified as Null Pointer Dereference that lead to "Denial of Service"
- A Null Pointer Dereference vulnerability could also potentially lead to code execution.



08
JUN
2017

## SAPCAR Heap Buffer Overflow: From crash to exploit

*BY: Maximiliano Vidal* *Latest from CoreLabs*

### 1. Introduction

CORELabs

In this blog post, we will cover the analysis and exploitation of a simple heap buffer overflow found in SAPCAR a few weeks ago.

SAP published security note #2441560 classifying the issue as "Potential Denial of Service". This post is our attempt to show that code execution is not only possible but also relatively easy to achieve. The idea is to provide a (hopefully!) cohesive example for other beginners that might be interested in binary exploitation. We will see one possible approach to make sense out of a few hundred crashes obtained through *fuzzing*, how to identify the root cause of the bug, and how to determine its exploitability. Afterwards, we will develop an exploit using the old and well known file pointer overwrite technique. The last section will go into some more detail about a relevant mitigation implemented in *glibc 2.24*.

# Common Vulnerabilities

- Sometime a crash report is rejected because it seems to be not exploitable

## FROM CRASH TO EXPLOIT: CVE-2015-6086 – OUT OF BOUND READ/ASLR BYPASS

18/01/2016 / 5 Comments on From Crash to Exploit: CVE-2015-6086 – Out of Bound Read/ASLR Bypass / in Research / by admin_payatu

### INTRODUCTION

This is a story of an `Out of Bound Read` bug in `Internet Explorer 9-11`. This is almost **5 years old bug** which got discovered in **April 2015**. It is a very interesting bug, at least from my perspective, because it was `rejected` almost **4-5 times** by **Zero Day Initiative (ZDI)** stating that it's `not exploitable`.

My **SVG fuzzer** was hitting a crash continuously, at first, the bug looked like usual `Use after Free` as it was trying to `read Invalid Memory`. But after triaging it turned out to be `Out of Bound Read` bug. I submitted this bug to **ZDI** and they rejected it at first stating that they are not able to reproduce it. Later they rejected

- For example if a vulnerability is reported as Denial of Service, the patch may not provide the proper mitigation
- The same vulnerability might be able to give code execution if analyzed/exploited properly

# Exploitable crashes

# Exploitable crashes



Some crashes that are generally exploitable:

- Stack overflow
- Crash on Program Counter
- Crash on branch
- Crash on write memory
- Heap vulnerabilities
- Read access violation ??

## Stack overflow (1/2)

When a stack buffer overflow occurs – e.g., overwriting control information such as a return address - libc (depending on the *gcc* version) generates a signal and stops the target:

```
*** stack smashing detected ***: ./stack_overflow terminated
Aborted (core dumped)
```

Generally this type of stack buffer overflow is considered exploitable – e.g., the size of the buffer is fixed and there is weak bounds checking

**How is it possible to detect?**

*__stack_chk_fail* terminates a function in case of stack overflow:

```
[0x7fea3e5ee428]> pd -1 @  sym.exploit_me+123
            0x0040060c      e85ffeffff     call sym.imp.__stack_chk_fail
```

## Stack overflow (2/2)

Look at the backtrace:

```
*** stack smashing detected ***: ./stack_overflow terminated
==13765==
==13765== Process terminating with default action of signal 6 (SIGABRT)
==13765==    at 0x4E6F428: raise (raise.c:54)
==13765==    by 0x4E71029: abort (abort.c:89)
==13765==    by 0x4EB17E9: __libc_message (libc_fatal.c:175)
==13765==    by 0x4F5315B: __fortify_fail (fortify_fail.c:37)
==13765==    by 0x4F530FF: __stack_chk_fail (stack_chk_fail.c:28)
==13765==    by 0x400610: exploit_me (in /home/invictus1306/Documents/r2conf/init_test/mytests/stack_overflow)
==13765==    by 0x400040: ??? (in /home/invictus1306/Documents/r2conf/init_test/mytests/stack_overflow)
==13765==    by 0xFFF0003F7: ???
==13765==    by 0xFFFFFFFF: ???
==13765==    by 0x40063F: ??? (in /home/invictus1306/Documents/r2conf/init_test/mytests/stack_overflow)
==13765==    by 0x4E5A82F: (below main) (libc-start.c:291)
==13765==
```

Looking at the backtrace it is possible to understand that the target program called the function *__stack_chk_*fail, which terminated it with the **SIGABRT** signal.

Another thing that is possible to see in the backtrace is the presence of the *__fortify_fail* function, it means that the StackGuard patch to *GCC* is there.

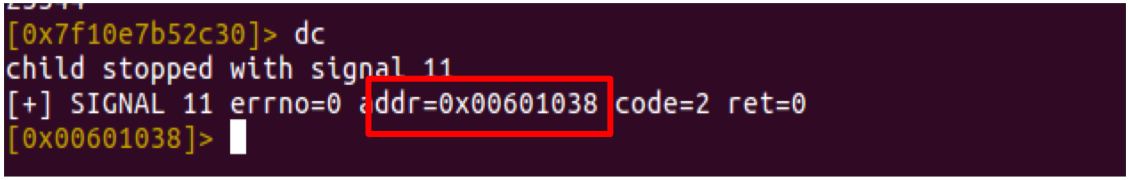
*gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.9)*

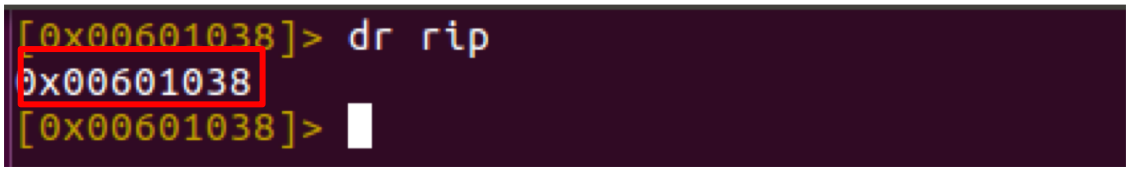
## Crash on Program Counter (1/2)

If there is a **SIGSEGV** signal and the address of the program counter is equal to the address of the memory error, this could tell us that the *PC* is tainted and can be controlled by an attacker.

Generally this type of crash is considered exploitable – e.g., the presence of a call instruction with a bad argument.

In the following case we have a **SIGSEGV** signal and we can see that the address of the "memory error" is equal to the "program counter" address.



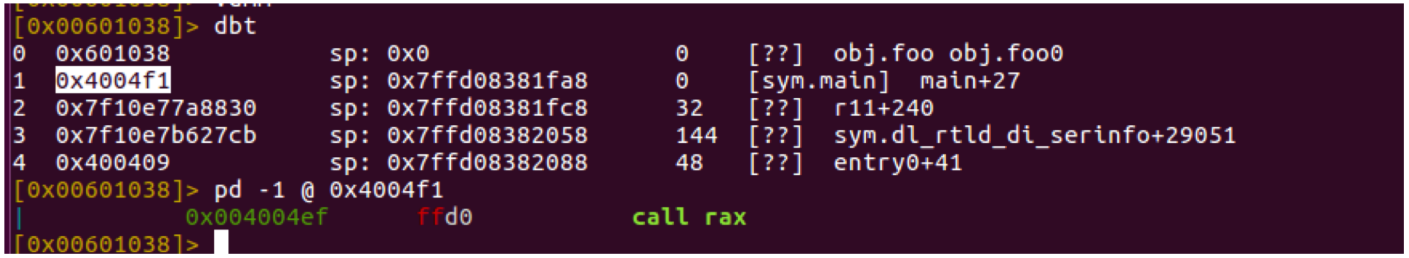




Let's go deeper…

## Crash on Program Counter (2/2)

Let's see the content of the current instruction:


Uhmm *NULL* bytes, but who is the caller?

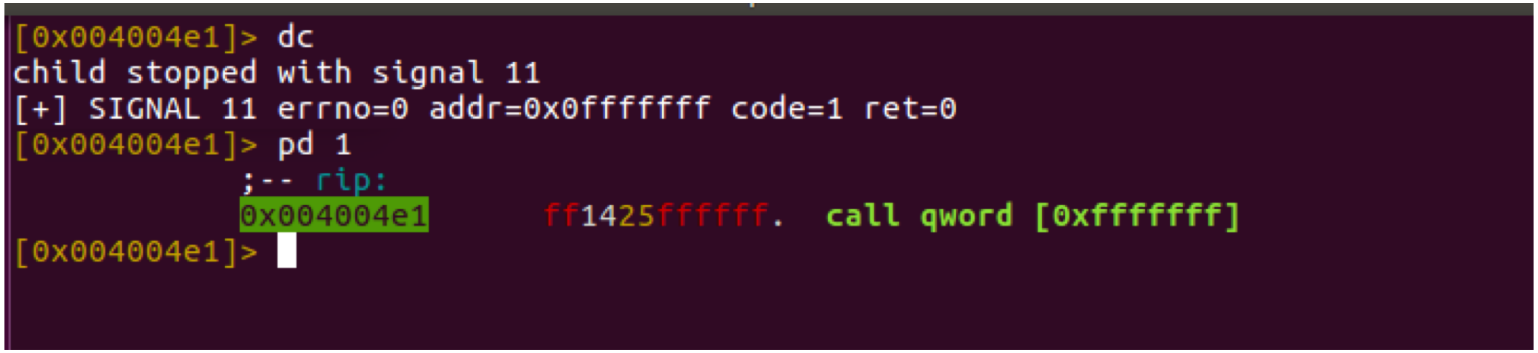



As we suspected there is a call statement, which has *rax* as parameter, this could mean that the *rax* value could be controlled by the attacker and as consequence it could be exploitable.

## Crash on Branch

A crash on a branch instruction could indicate that the control flow could be controlled by the attacker .

For example, in that case:




We have a **SIGSEGV** signal, and the current instruction is a call to an address that is not mapped in the memory.

## Crash on write memory

An access violation on a destination operand could indicate a write access violation.

```
[0x7f2f01f32c30]> dc
child stopped with signal 11
[+] SIGNAL 11 errno=0 addr=0x00010000 code=1 ret=0
[0x004004eb]> pd 1
            ;-- rip:
            0x004004eb      c60041          mov byte [rax], 0x41        ; 'A' ; [0x41:1]=255 ; 65
```

Generally this type of crash is considered exploitable – e.g., an attacker may control the write address/value.

Note: The faulting address is equal to the destination operand value:

*Faulting address:* **0x10000**
*Fault instruction:* **mov byte [rax], 0x41**
*rax* = **0x10000**

## Heap vulnerabilities

The newer version of *glibc* added a lot of sanity checks for chunk headers:

```
[#0] 0x7ffff7a42428 → Name: __GI_raise(sig=0x6)
[#1] 0x7ffff7a4402a → Name: __GI_abort()
[#2] 0x7ffff7a847ea → Name: __libc_message(do_abort=0x2, fmt=0x7ffff7b9ded8 "*** Error in `%s': %s: 0x%s ***\n")
[#3] 0x7ffff7a8d37a → Name: malloc_printerr(ar_ptr=<optimized out>, ptr=<optimized out>, str=0x7ffff7b9df50 "free(): invalid next size (fast)", action=0x3)
[#4] 0x7ffff7a8d37a → Name: _int_free(av=<optimized out>, p=<optimized out>, have_lock=0x0)
[#5] 0x7ffff7a9153c → Name: __GI___libc_free(mem=<optimized out>)
[#6] 0x4005f5 → Name: main()
```

```
[#0] 0x7ffff7a42428 → Name: __GI_raise(sig=0x6)
[#1] 0x7ffff7a4402a → Name: __GI_abort()
[#2] 0x7ffff7a847ea → Name: __libc_message(do_abort=0x2, fmt=0x7ffff7b9ded8 "*** Error in `%s': %s: 0x%s ***\n")
[#3] 0x7ffff7a8d37a → Name: malloc_printerr(ar_ptr=<optimized out>, ptr=<optimized out>, str=0x7ffff7b9dfa0 "double free or corruption (fasttop)", act
[#4] 0x7ffff7a8d37a → Name: _int_free(av=<optimized out>, p=<optimized out>, have_lock=0x0)
[#5] 0x7ffff7a9153c → Name: __GI___libc_free(mem=<optimized out>)
[#6] 0x400762 → Name: main()
```

Looking at the backtrace it is possible to see the target program calls the function *_int_free,* and also see the *malloc_printerr* function contains the descriptive string.

This helps the analyzer to understand the type of vulnerability.

# Exploitable crashes

## Read access violation that leads to code execution

As mentioned, sometimes a simple read access violation vulnerability could lead to code execution.

```
   0x401bfd <main+1177>      mov    rdi, rax
   0x401c00 <main+1180>      call   0x401dbc <_ZN4Base3setEff>
   0x401c05 <main+1185>      mov    rax, QWORD PTR [rbp-0x68]
   0x401c09 <main+1189>      mov    rax, QWORD PTR [rax]
   0x401c0c <main+1192>      add    rax, 0x8
→  0x401c10 <main+1196>      mov    rax, QWORD PTR [rax]
   0x401c13 <main+1199>      mov    rdx, QWORD PTR [rbp-0x68]
   0x401c17 <main+1203>      mov    rdi, rdx
   0x401c1a <main+1206>      call   rax
   0x401c1c <main+1208>      movd   eax, xmm0
   0x401c20 <main+1212>      mov    DWORD PTR [rbp-0x84], eax
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000401c10 in main ()

$rax   : 0x0000000000000008
$rbx   : 0x000000000061 6c80  →  0x0000000000000000
```

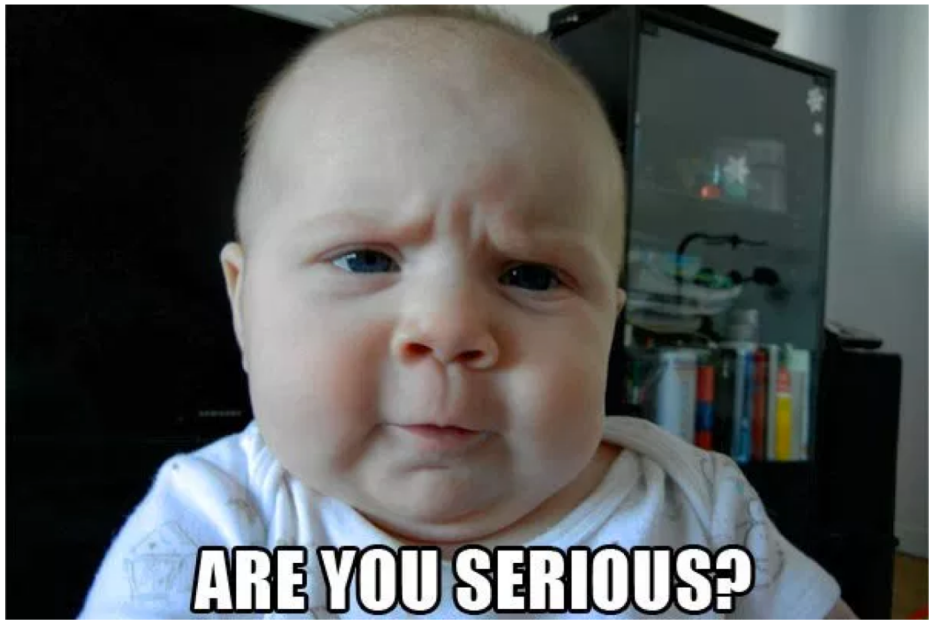
The "exploitable" gdb plugin tell us the crash is PROBABLY_NOT_EXPLOITABLE

```
gef➤  exploitable
__main__:99: UserWarning: GDB v7.11 may not support required Python API
Description: Access violation near NULL on source operand
Short description: SourceAvNearNull (16/22)
Hash: c0f280e80c376bd06c228bee 621e541.c0f280e80c376bd06 228bee2621e541
Exploitability Classification: PROBABLY_NOT_EXPLOITABLE
Explanation: The target crashe  an address matching the source operand of the current instruction. This likely indi
cates a read access violation, which may mean the application crashed on a simple NULL dereference to data structure that has no immediate ef
fect on control of the processor.
Other tags: AccessViolation (21/22)
```

## Read access violation that leads to code execution

Instead this is exploitable, in this case the *rax* register is controllable from an attacker.

```
[0x00401c10]> pd 10
            0x00401c10      488b00          mov rax, qword [rax]
            0x00401c13      488b5598        mov rdx, qword [rbp - 0x68]
            0x00401c17      4889d7          mov rdi, rdx
            0x00401c1a      ffd0            call rax
```



# Tool and usage

# Tool and usage



New tool released as open source:

**beebug**

***https://github.com/invictus1306/beebug***

Some features:

- Classify if a crash could be exploitable:
  - Stack overflow on libc
  - Crash on Program Counter
  - Crash on branch
  - Crash on write memory
  - Heap vulnerabilities
  - Read access violation (some exploitable cases)
- Help to analyze a crash (graph view)

# Tool and usage

CON 2018

## Help view

```
root@invictus1306-VirtualBox:/home/invictus1306/Documents/r2conf/beebug/beebug# python3 beebug.py -h
usage: beebug.py [-h] -t TARGET [-a TARGETARGS] [-f FILE] [-g GRAPH]

optional arguments:
  -h, --help            show this help message and exit
  -t TARGET, --target TARGET
                        target program to analyze
  -a TARGETARGS, --targetargs TARGETARGS
                        arguments for the target program
  -f FILE, --file FILE  input file
  -g GRAPH, --graph GRAPH
                        generate the graph
```

## Output example

```
Crash on PC - Generally it is exploitable, the PC could be tainted
backtrace
0  0x601038           sp: 0x0                 0    [??]  obj.foo obj.foo0
1  0x4004f1           sp: 0x7fff447da688      0    [sym.main]  main+27
2  0x7fba9cd9f830     sp: 0x7fff447da6a8      32   [??]  r11+240
3  0x7fba9d1597cb     sp: 0x7fff447da738      144  [??]  sym.dl_rtld_di_serinfo+29051
4  0x400409           sp: 0x7fff447da768      48   [??]  entry0+41
registers
rax = 0x00601038
rbx = 0x00000000
rcx = 0x00000000
rdx = 0x7fff447da798
r8 = 0x00400570
r9 = 0x7fba9d159ab0
r10 = 0x00000846
r11 = 0x7fba9cd9f740
r12 = 0x004003e0
r13 = 0x7fff447da780
r14 = 0x00000000
```

# Tool and usage

- Example with **CVE-2018-12327** (ntpq 4.2.8p11 Local Buffer Overflow)

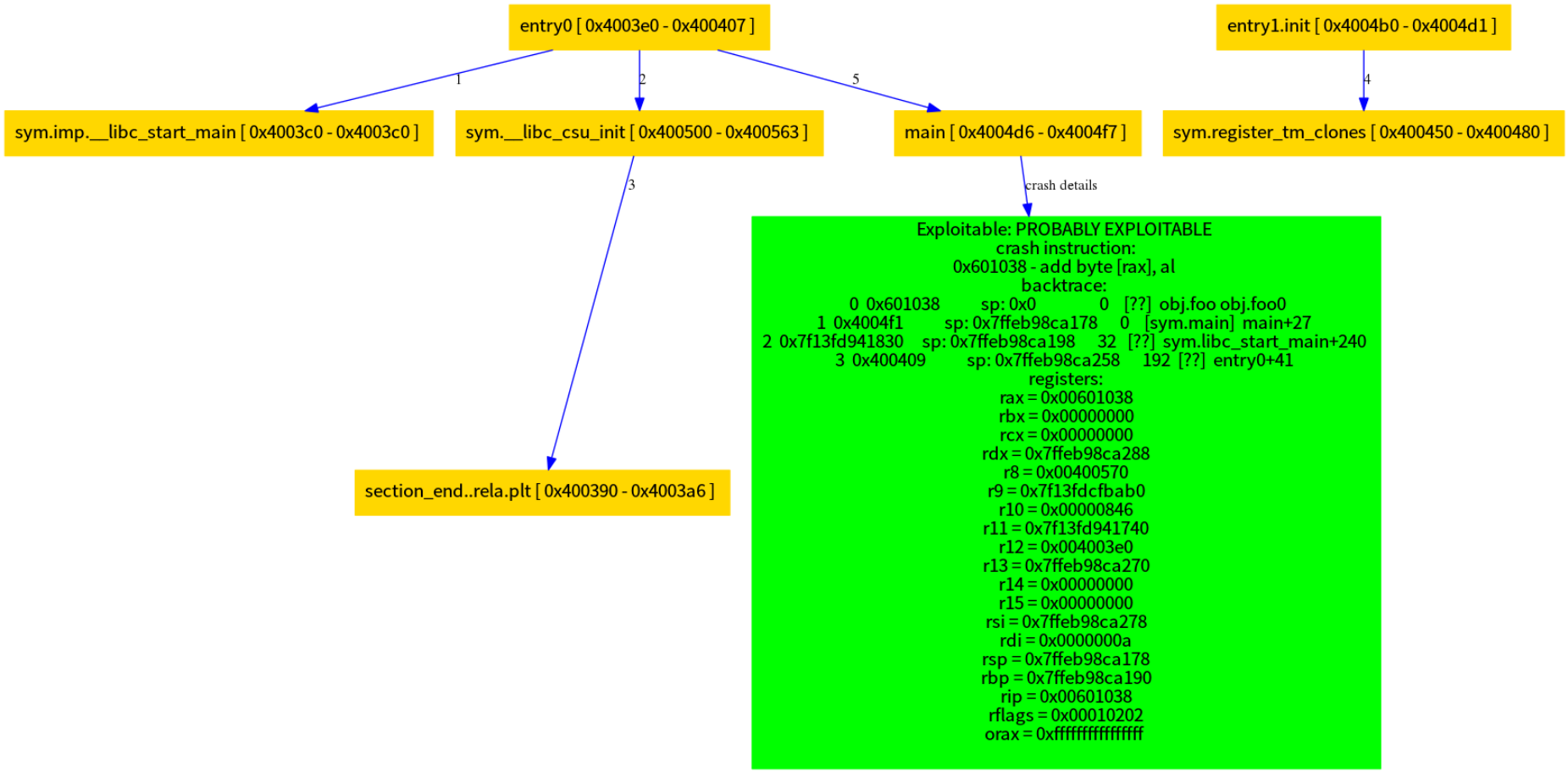
```
root@invictus1306-VirtualBox:/home/invictus1306/Documents/r2conf/beebug/beebug# python3 beebug.py -t /home/invictus1306/Do
wnloads/ntp-4.2.8p11/ntpdc/ntpdc -a "-4 [`python -c 'print "A" * 300'`]"
Process with PID 5017 started...
File dbg:///home/invictus1306/Downloads/ntp-4.2.8p11/ntpdc/ntpdc -4 [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA] reopened in read-write mode
= attach 5017 5017
WARNING: bin_strings buffer is too big (0xffffaa1e5556a5f8). Use -zzz or set bin.maxstrbuf (RABIN2_MAXSTRBUF) in r2 (rabin
2)
WARNING: bin_strings buffer is too big (0xffffaa1e5555a258). Use -zzz or set bin.maxstrbuf (RABIN2_MAXSTRBUF) in r2 (rabin
2)
Cannot find function 'entry0' at 0x00007bd0
Name or service not known
*** stack smashing detected ***: /home/invictus1306/Downloads/ntp-4.2.8p11/ntpdc/ntpdc terminated
child stopped with signal 6
[+] SIGNAL 6 errno=0 addr=0x00001399 code=-6 ret=0
Stack error - Generally it could be exploitable, due to suspicious functions in the backtrace
backtrace
0  0x7fc8112a8428     sp: 0x0                 0    [??]
1  0x7fc8112aa02a     sp: 0x7ffcbfab54e8      0    [??]  sym.abort+362
2  0x7fc8112ea7ea     sp: 0x7ffcbfab5618      304  [??]  sym.fsetlocking+778
3  0x7fc8112ea5ce     sp: 0x7ffcbfab56a8      144  [??]  sym.fsetlocking+238
```

# Graph generation

# Graph generation

- As already mentioned, some crashes are not easy to classify
- With **beebug** there is the possibility to have a runtime graph view
- The sequence of the executed functions will be printed out
- This could help the analyzer to have a better understanding of what is happening
- For every node in the graph this is the label:
  - Function name [start_address – end_address]
  - The edge, contains the number in the sequence of execution

# Graph generation



# DEMO

# Future direction

# Future direction



- Support different architectures
- Improvement of the graph view (based on radare2)
- Analyze core dumps (based on radare2)
- Use instrumentation for the graph view generation

**beebug**

***https://github.com/invictus1306/beebug***

# Thank you!

Andrea Sindoni

  @invictus1306