Leveraging CVE-2015-7547 for ASLR Bypass & RCE

Gal De Leon & Nadav Markus
Who We Are

• Nadav Markus, Gal De-Leon
• Security researchers @ PaloAltoNetworks
  • Vulnerability research and exploitation
  • Reverse engineering
• Traps exploits mitigation research
• Security enthusiasts
Introduction

- Vulnerability in glibc
- Discovered by Google, patched at 16-feb-2016
  - Released crash-poc
- This presentation is about exploitation strategy
getaddrinfo()

- A function used to resolve a hostname to IP-address(es)
  - OUT-addressinfo* is later used for connect() or bind()

- Performs DNS queries

- Stack-overflow vulnerability was found, handling DNS-replies

“Given node and service, which identify an Internet host and a service, getaddrinfo() returns one or more addrinfo structures, each of which contains an Internet address that can be specified in a call to bind(2) or connect(2).”
getaddrinfo()

- struct addrinfo -> ai_family
  - AF_INET, AF_INET6, **AF_UNSPEC**
  - AF_UNSPEC indicates any address family is valid, IPv4 or IPv6
    - IPv4=A, IPv6=AAAA
  - This is the common usecase

```c
int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);
```
The Bug

calls glibc’s getaddrinfo()
alloca(2048)

```c
_nss_dns_gethostbyname4_r(..) {
    ...
    host_buffer.buf = orig_host_buffer = (querybuf *) alloca(2048);
```
The Bug

Victim

Attacker

dns-reply FLAGS=TRUNCATED, ‘\xAA’*2500

_alloca() buffer isn’t sufficient
malloc(MAXPACKET) instead
(MAXPACKET=0x10000)

“...If the answer section of the response is truncated and if the
requester supports TCP, it SHOULD try the query again using
TCP.” (RFC-DNS)
**The Bug**

dns-query A & AAAA over TCP
The Bug

reuses the ‘malloced’ buffer
The Bug

Mismatch between the stack and the heap buffer!

If ‘AAAA’ dns-response’s size is greater than 2048, we smash the stack!
**DEP & ASLR**

- Unable to exploit reliably without bypassing DEP & ASLR
  - We can set RIP to anywhere, but where to?

- How to ROP?

```
Overview
.
.
RET ; pop, jmp
```
ASLR Bypass Techniques

- Non PIE executables
- Read memory vulnerability
- Spray (without DEP)
  - Can’t get reliable address at 64 bit..
- How about guessing?
fork()

- Standard way of creating processes in UNIX
- Child process inherits:
  - Memory Layout (includes loaded modules 😊)
  - Registers state
  - Stack

```c
pid = fork();

if (0 < pid) {
    /* Parent code goes here... */
} else if (0 == pid) {
    /* Child code goes here... */
}
```

Reply Arbitrary DNS

- Assume attacker can answer arbitrary DNS requests
- Acquiring the domain
- Local Arp Poisoning
- Any other way ..
Attacker 10.1.1.137

fork()

Victim

child-process (PID=3312):
Attacker 10.1.1.137

fork()

Victim

child-process (PID=3312):
1212100C CALL getaddrinfo
12121012 test eax, eax
...

dns-queries
Attacker 10.1.1.137

fork()

Victim

child-process (PID=3312):
1212100C CALL getaddrinfo
OVERFLOW RIP=0x01010101
INVALID-OPCODE
CRASH!
12121012 test eax, eax
...

malicious dns-replies host @ 10.1.1.137

Attacker
10.1.1.137
Attacker
10.1.1.137

Victim

Trigger fork()
Attacker 10.1.1.137

fork()

Victim

call-process (PID=3313):
1212100C CALL getaddrinfo
12121012 test eax, eax
...

dns-queries

Attacker
10.1.1.137
Attacker 10.1.1.137

fork()

Victim

child-process (PID=3313):
1212100C CALL getaddrinfo
OVERFLOW RIP=0x12121012
12121012 test eax, eax
...

malicious dns-replies host @ 10.1.1.137

Attacker 10.1.1.137
Attacker 10.1.1.137

fork()

Victim

child-process (PID=3313):
1212100C CALL getaddrinfo
OVERFLOW RIP=0x12121012
12121012 test eax, eax
... connect()

TCP syn 10.1.1.137

0x12121012 is the correct address!

Attacker 10.1.1.137
Exploit Strategy

- We can enumerate all possible addresses
- \( \sim 2^{64} \)
  - Not feasible
**Byte by Byte Approach**

- Instead of overwriting RIP entirely, we can overwrite just a portion of it.
- Remaining bytes are of the original address.

```
CALL getaddrinfo()
Original RIP = 0x12121012
```

```
Overwritten RIP = 0x12121000
```
## Byte by Byte Approach

<table>
<thead>
<tr>
<th>Return Address</th>
<th>Sent Buffer</th>
<th>Response?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000012121000</td>
<td>AAA...0x00</td>
<td>NO</td>
</tr>
<tr>
<td>0x000000012121001</td>
<td>AAA...0x01</td>
<td>NO</td>
</tr>
<tr>
<td>0x0000000121210...</td>
<td>...</td>
<td>NO</td>
</tr>
<tr>
<td>0x000000012121012</td>
<td>AAA...0x12</td>
<td>YES</td>
</tr>
<tr>
<td>0x000000012120012</td>
<td>AAA...0x12 0x00</td>
<td>NO</td>
</tr>
<tr>
<td>0x000000012120112</td>
<td>AAA...0x12 0x01</td>
<td>NO</td>
</tr>
<tr>
<td>0x00000001212.....12</td>
<td>...</td>
<td>NO</td>
</tr>
<tr>
<td>0x000000012121012</td>
<td>AAA...0x12 0x10</td>
<td>YES</td>
</tr>
<tr>
<td>.... ...</td>
<td>8 * 256 ☺</td>
<td></td>
</tr>
</tbody>
</table>
Finding Potentially Exploitable Applications

- fork() && getaddrinfo()
  - Using the correct flow
- http://codesearch.debian.net
  - Indexes source code of ~18,000 packages
- ~1,300 potential exploitable apps
FindingPotentiallyExploitableApplications

...
914 xtrace
915 openbgpd
916 balsa
917 tinyproxy
918 powerman
919 mahimahi
920 pcs
921 eric
922 ruby-pg
923 nut
924 gnulib
...

Finding Potentially Exploitable Applications

...  
914 xtrace  
915 openbgpd  
916 balsa  
917 **tinyproxy**  
918 powerman  
919 mahimahi  
920 pcs  
921 eric  
922 ruby-pg  
923 nut  
924 gnulib  
...
1. CONNECT is used to trigger fork() remotely
2. http-request is used as an indication of success
Game Over?
Is RIP Really Controlled?

- So let’s enumerate getaddrinfo’s return address
- But we crash 😞
- Local-variables are overridden
  - Some are pointers ..
  - So let’s leak them 😊

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack buffer</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Local variable</td>
</tr>
<tr>
<td>Local variable</td>
</tr>
<tr>
<td>Local variable</td>
</tr>
<tr>
<td>Local variable</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Frame pointer</td>
</tr>
<tr>
<td>Return address</td>
</tr>
</tbody>
</table>
**Segfault (1st)**

- RBX originally points the stack
- We can leak this address too!
- Address pointed by RBX *only* has to be writeable
  - Flow is not effected
Leak RBX (arbitrary stack address)

(/proc/PID/maps)

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7ffd07882000</td>
<td>0x7ffd078a3000</td>
<td>0x21000</td>
</tr>
<tr>
<td>0x7ffd079f0000</td>
<td>0x7ffd079f2000</td>
<td>0x2000</td>
</tr>
</tbody>
</table>

• We cannot leak lower 12 bits, since anything will do 😊
  • Stack size is greater than 0x1000

| AA | A2 | 88 | 07 | FD | 7F | 00 | 00 |

0x7ffd078a3000

Stack

0x7ffd07882000

mov [rbx], sil
add rbx, 1
jmp loc_7F0E3D7A3A2B
**Segfault (2\textsuperscript{nd})**

- RDI is controlled (hostbuffer.buf)
- R14 is the original alloca() buffer
- free(RDI) if not equal
  - Abort crash
  - We don’t get to overwrite RIP ..
- How to predict R14 value?

```c
host_buffer.buf = orig_host_buffer = (querybuf *) alloca (2048);
... /* might malloc host_buffer.buf */
if (host_buffer.buf != orig_host_buffer)
    free (host_buffer.buf);
... return status;
```
Stack is Identical for Different fork()s

```
orig_host_buffer = alloca(2048);
orig_host_buffer @ 0x7fffd078a3080
```

```
orig_host_buffer = alloca(2048);
orig_host_buffer @ 0x7fffd078a3080
```

```
orig_host_buffer = alloca(2048);
orig_host_buffer @ 0x7fffd078a3080
```
**Precise Pointers**

- Stack is identical in different forks
  - `host_buffer.buf` PTR is the same in all executions
- Use constant offset from stack base!

```c
... host_buffer.buf = orig_host_buffer = (querybuf *) alloca (2048);
... /* might malloc host_buffer.buf */
if (host_buffer.buf != orig_host_buffer)
    free (host_buffer.buf);
...
return status;
```
Leak Stack Base

- Use 1th crash (mov [rbx], sil)
  - Use leaked arbitrary stack address as a starting point
- Add 0x1000 offset at a time

<table>
<thead>
<tr>
<th>rbx</th>
<th>Sent Buffer</th>
<th>Response?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00007ffffffd00000</td>
<td>AAA...0x00 0x00 0x00 0xed 0xff 0x7f</td>
<td>YES</td>
</tr>
<tr>
<td>0x00007ffffffd01000</td>
<td>AAA...0x00 0x10 0x00 0xed 0xff 0x7f</td>
<td>YES</td>
</tr>
<tr>
<td>0x00007ffffffd02000</td>
<td>AAA...0x00 0x20 0x00 0xed 0xff 0x7f</td>
<td>YES</td>
</tr>
<tr>
<td>0x00007ffffffd03000</td>
<td>AAA...0x00 0x30 0x00 0xed 0xff 0x7f</td>
<td>YES</td>
</tr>
<tr>
<td>0x00007ffffffd04000</td>
<td>AAA...0x00 0x40 0x00 0xed 0xff 0x7f</td>
<td>YES</td>
</tr>
<tr>
<td>0x00007ffffffd05000</td>
<td>AAA...0x00 0x50 0x00 0xed 0xff 0x7f</td>
<td>YES</td>
</tr>
<tr>
<td>0x00007ffffffd06000</td>
<td>AAA...0x00 0x60 0x00 0xed 0xff 0x7f</td>
<td>YES</td>
</tr>
<tr>
<td>0x00007ffffffd07000</td>
<td>AAA...0x00 0x70 0x00 0xed 0xff 0x7f</td>
<td>YES</td>
</tr>
<tr>
<td>0x00007ffffffd08000</td>
<td>AAA...0x00 0x80 0x00 0xed 0xff 0x7f</td>
<td>NO</td>
</tr>
</tbody>
</table>
### Game Over?

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xaaa0000</td>
<td>Stack base</td>
</tr>
<tr>
<td>0xa9fc00 (-0x400)</td>
<td>orig_host_buffer</td>
</tr>
<tr>
<td>0xa9f500 (-0xb00)</td>
<td>Local variable</td>
</tr>
<tr>
<td>0xa9f4f8 (-0xb08)</td>
<td>Local variable</td>
</tr>
<tr>
<td>0xa9f400 (-0xc00)</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>0xa9f3f8 (-0xc08)</td>
<td>Return address</td>
</tr>
</tbody>
</table>

Leaked =>

0x400 is constant offset =>
Offset from Stack-Base is Constant?

/include/linux/mm.h

unsigned long arch_align_stack(unsigned long sp)
{
    if (!((current->personality & ADDR_NO_RANDOMIZE)
        && randomize_va_space))
        sp -= get_random_int() % 8192;
    return sp & ~0xf;
}
Leak libc base (for fun & gadgets)

- Gain control over RIP (leak all local variables in the way ...)
- Leak getaddrinfo()'s return address
- Get libc's base address
  - Constant offset from ret address!
- ROP (ret2libc)
**libc Version?**

- Different offset for different versions
- We don’t know what the version is
- So we just enumerate 😊
Complete Exploitation Flow

• Leak arbitrary stack pointer (1st segfault => rbx)
• Leak stack base
• Leak random stack offset (for precise stack variables)
• Leak getaddrinfo()’s return address
• Enumerate ret2libc offsets, until successfully exploited
Demo
Conclusion

• Security Mitigations

• Bypass by abusing OS features (fork syscall)

• This technique can be used with other server vulnerabilities

• How to protect?
  • Use Palo Alto Networks security platform
  • Patch libc!
Questions?