Leveraging CVE-2015-7547 for ASLR Bypass & RCE

Gal De Leon & Nadav Markus

paloalto NETWORKS®

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

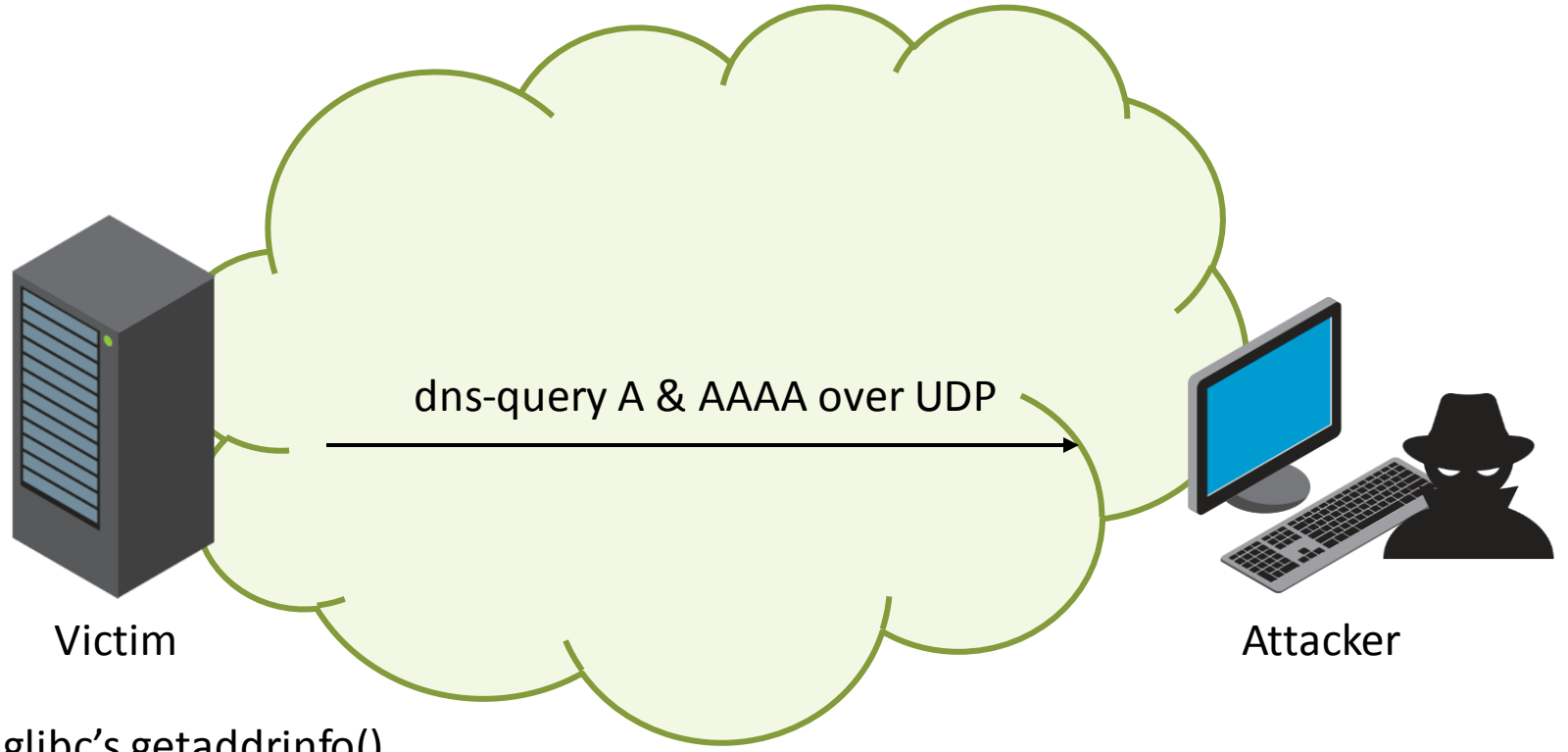
"Given *node* and *service*, which identify an Internet host and a service, **getaddrinfo**() returns one or more *addrinfo* structures, each of which contains an Internet address that can be specified in a call to **bind**(2) or **connect**(2)."

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

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

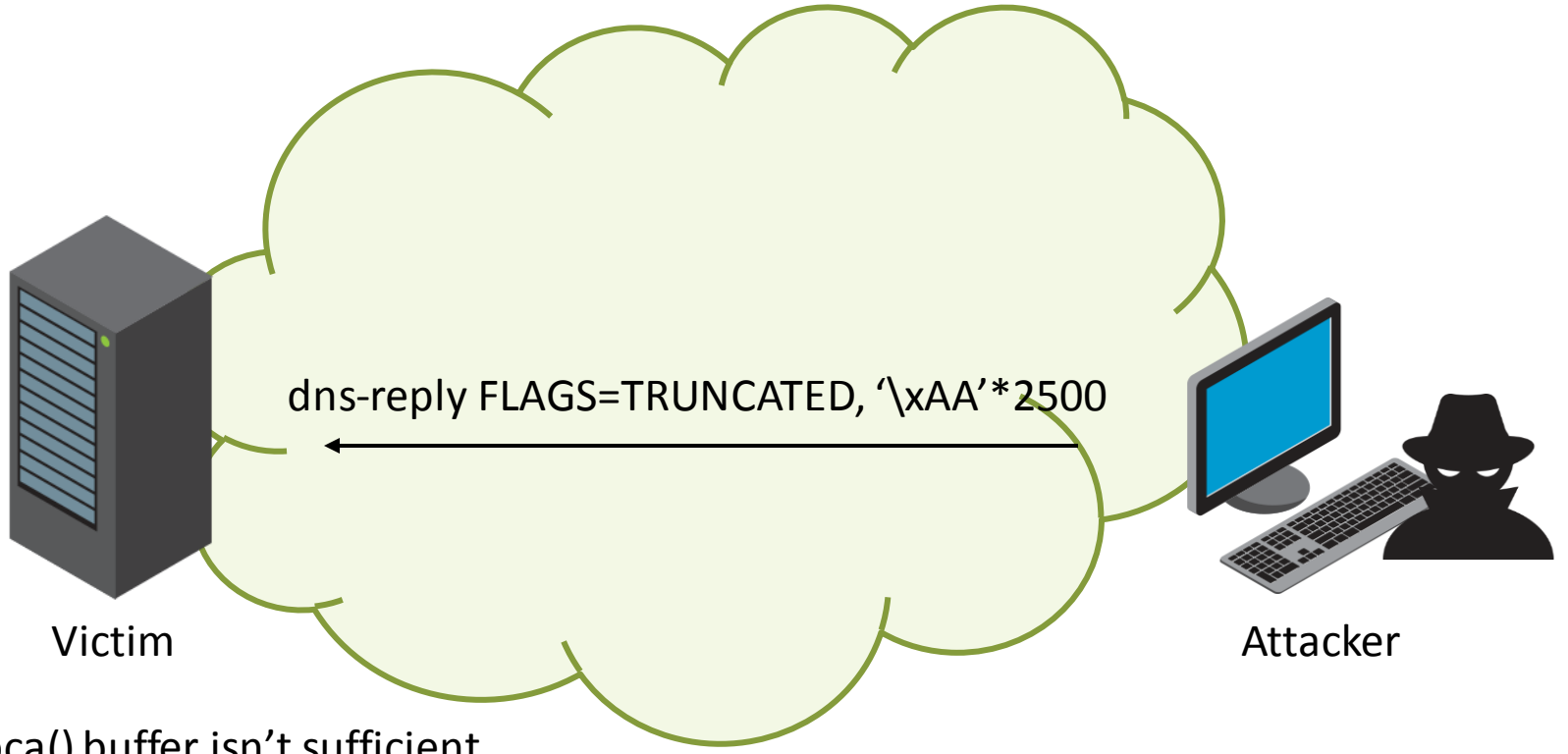
The Bug



calls glibc's getaddrinfo()
alloca(2048)

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

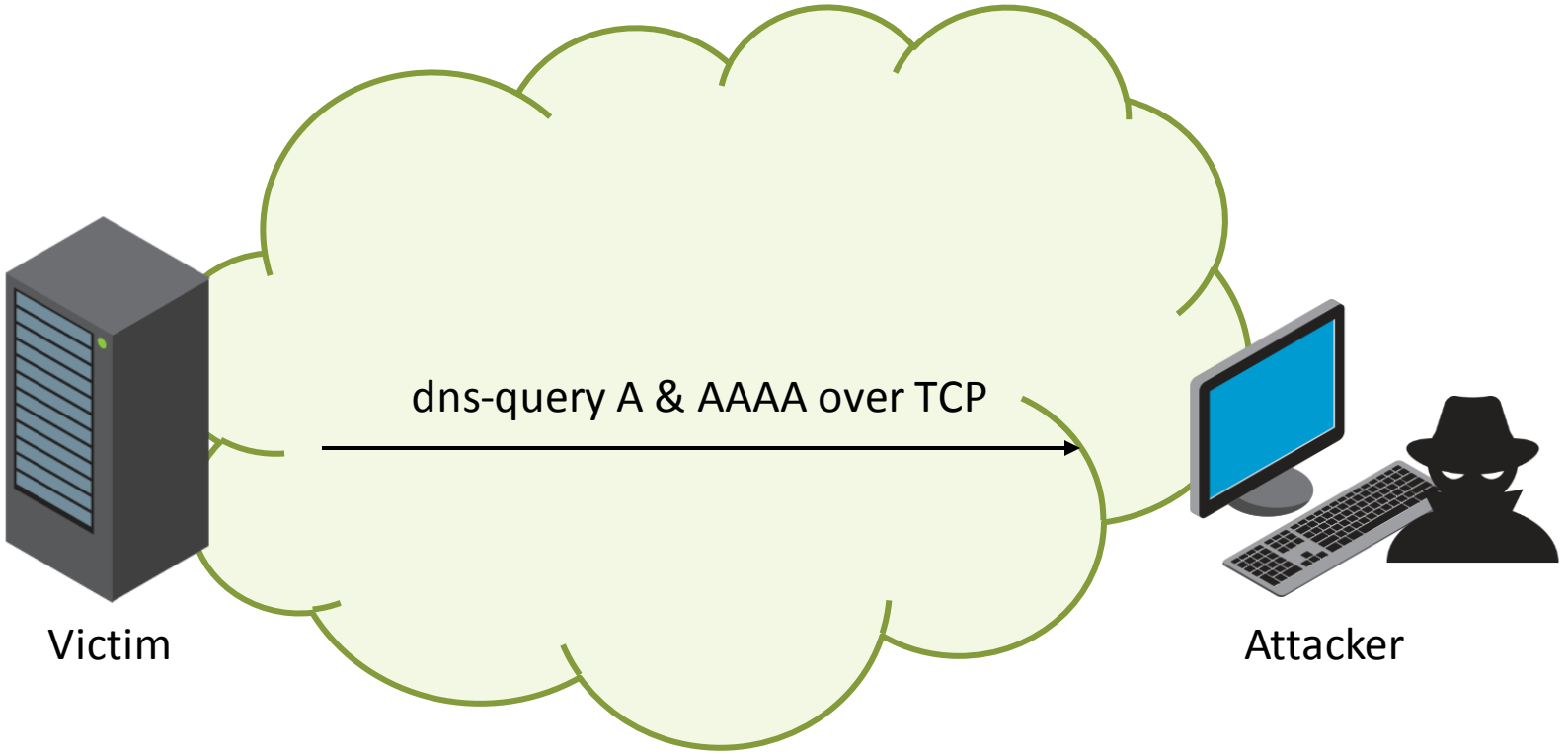
The Bug



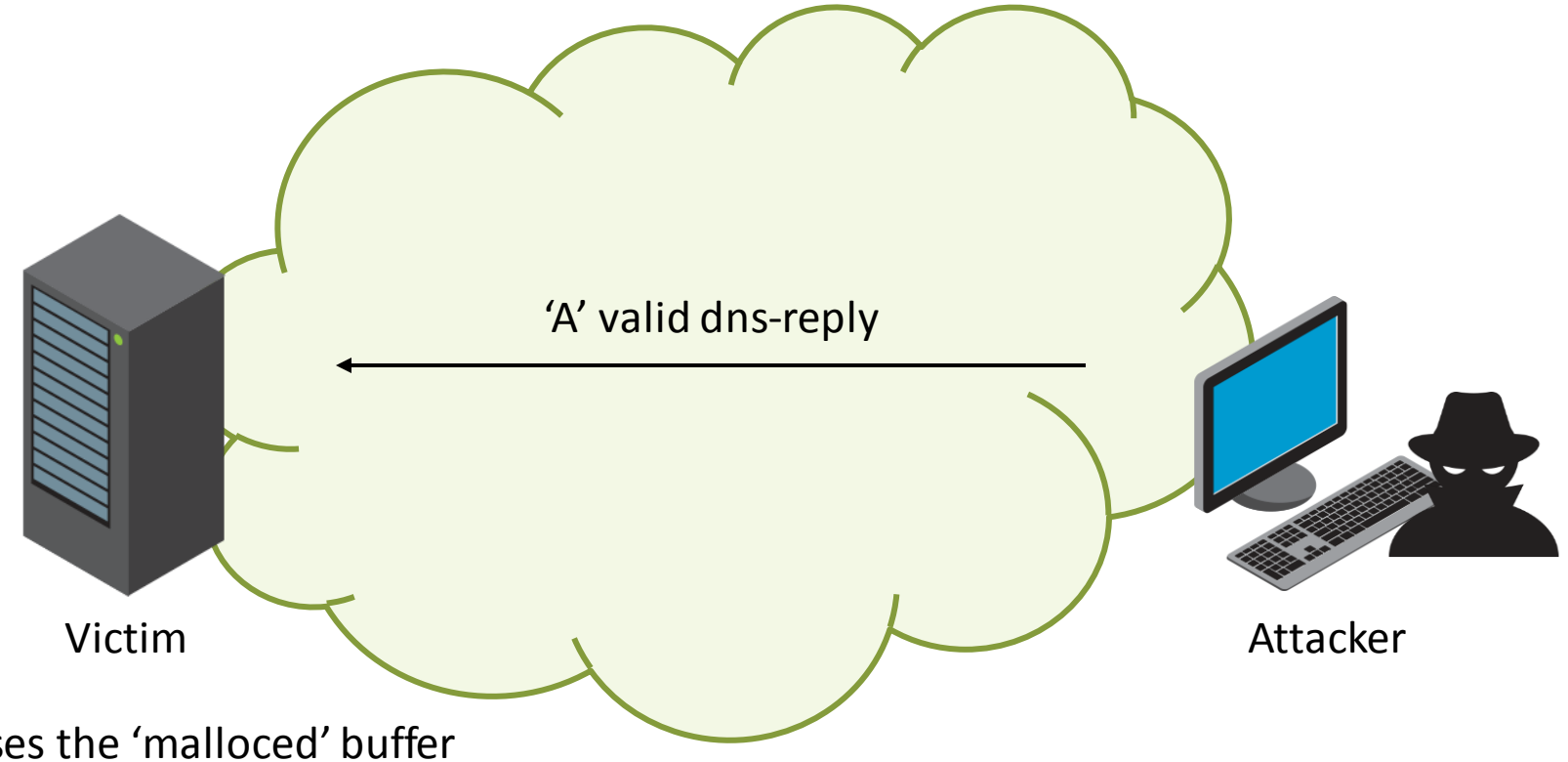
_alloca() buffer isn't sufficient
malloc(MAXPACKET) instead
(MAXPACKET=0x10000)

"…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

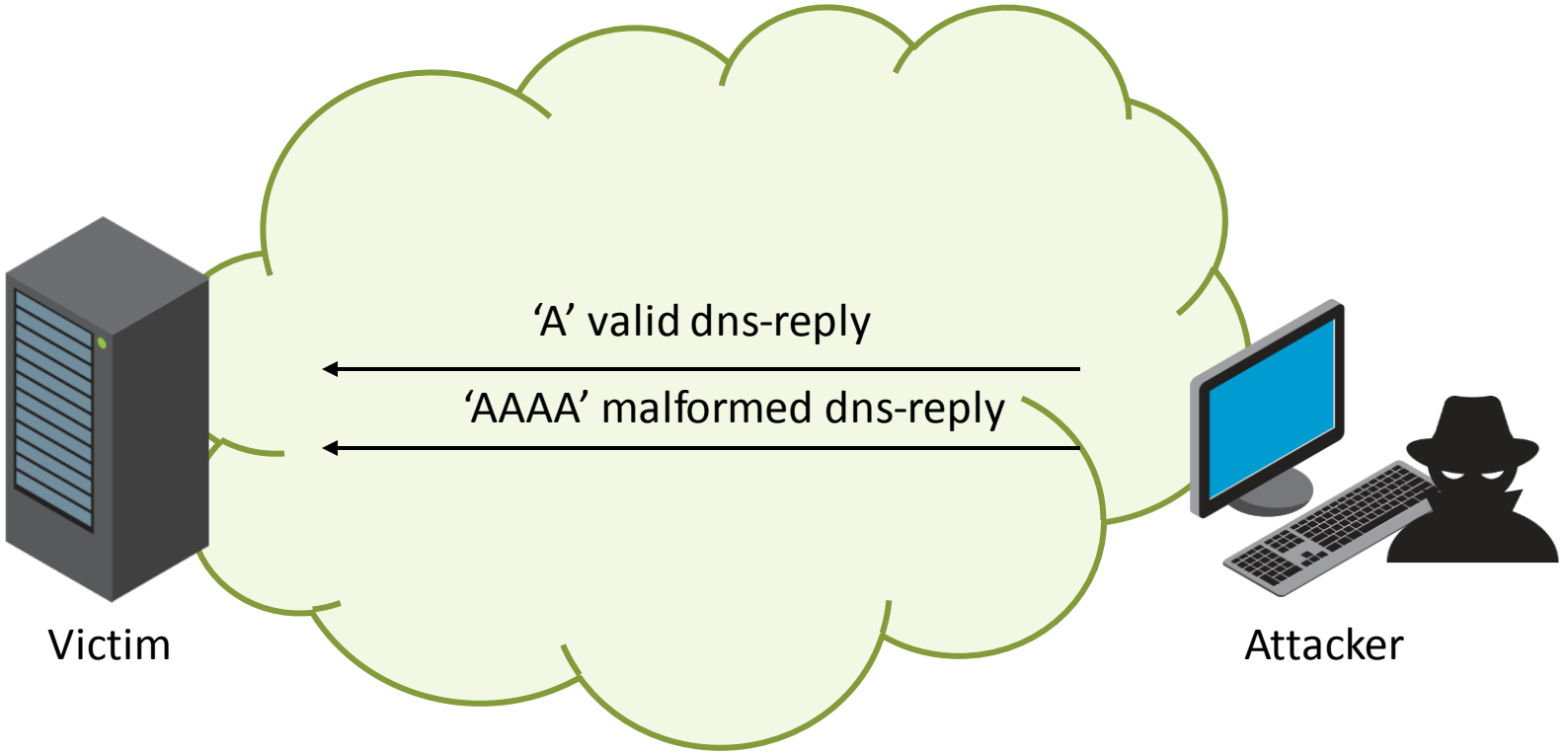


The Bug



reuses the 'malloced' buffer

The Bug

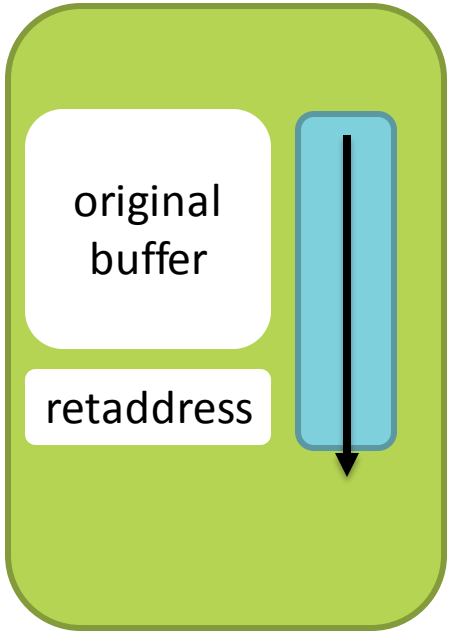


Mismatch between the stack and the heap buffer!

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?



Overflow
.
.
.
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?

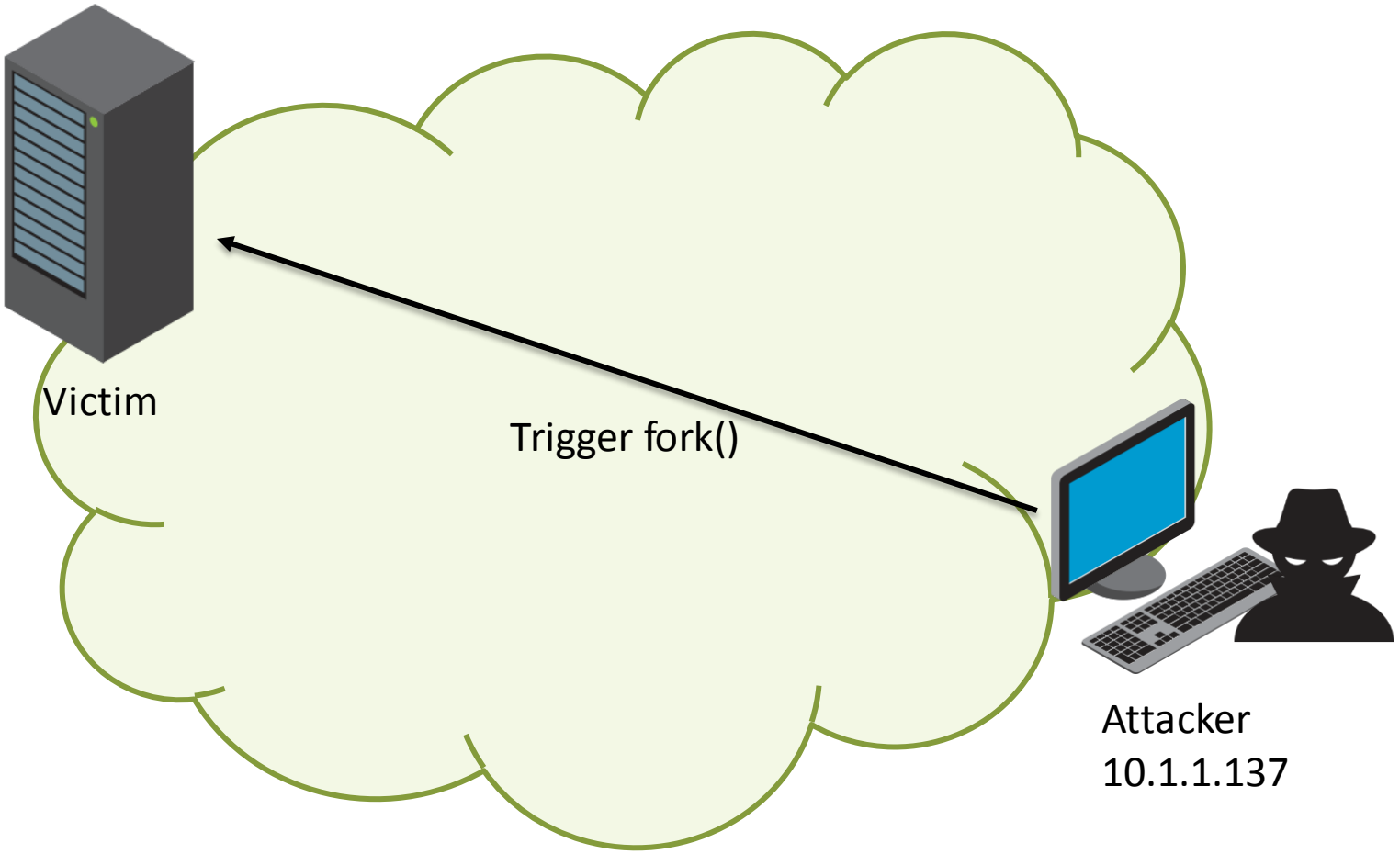
fork()

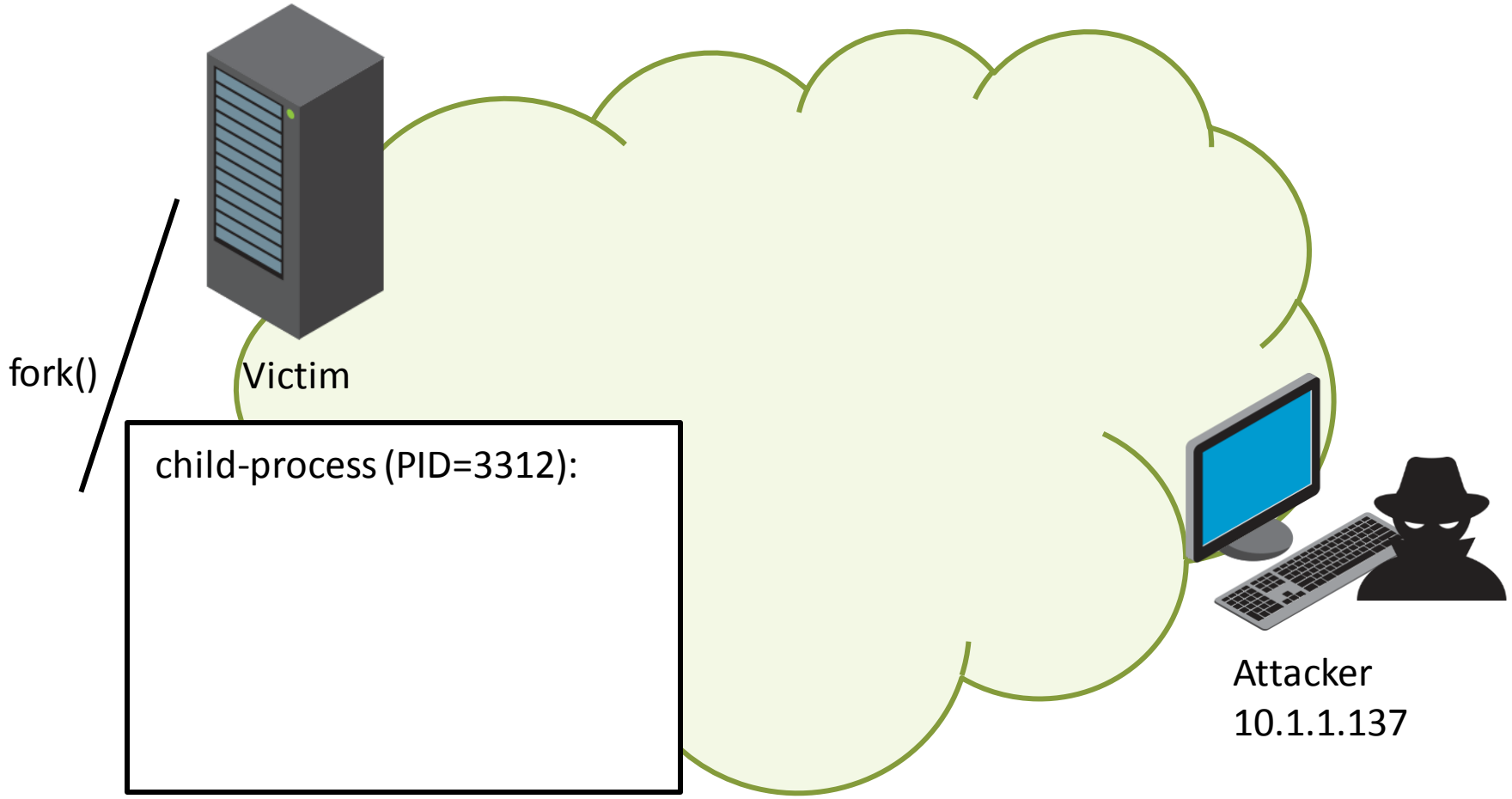
- Standard way of creating processes in UNIX
- Child process inherits:
  - Memory Layout (includes loaded modules ☺)
  - Registers state
  - Stack

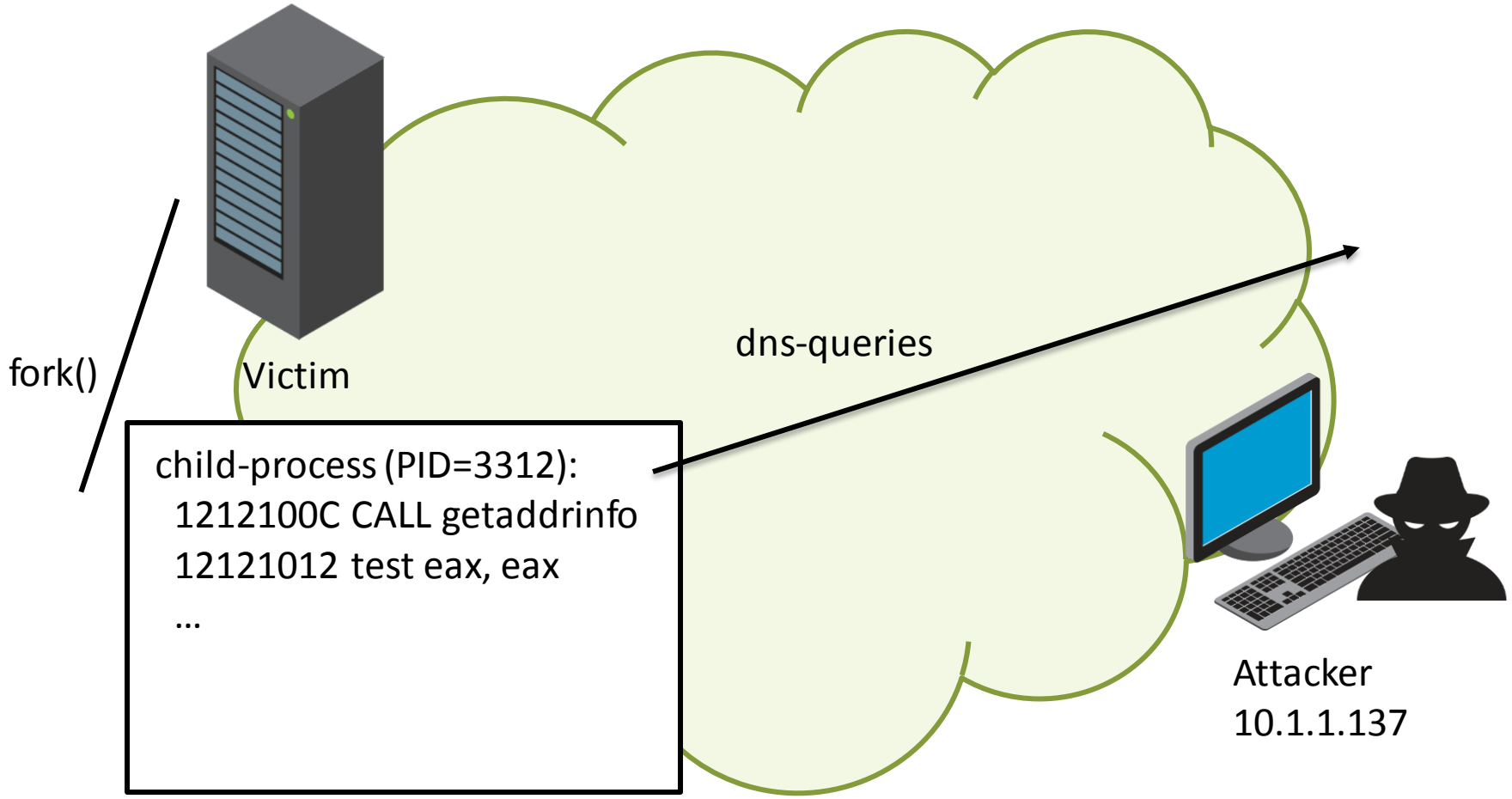
```
pid = fork();

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

...
```

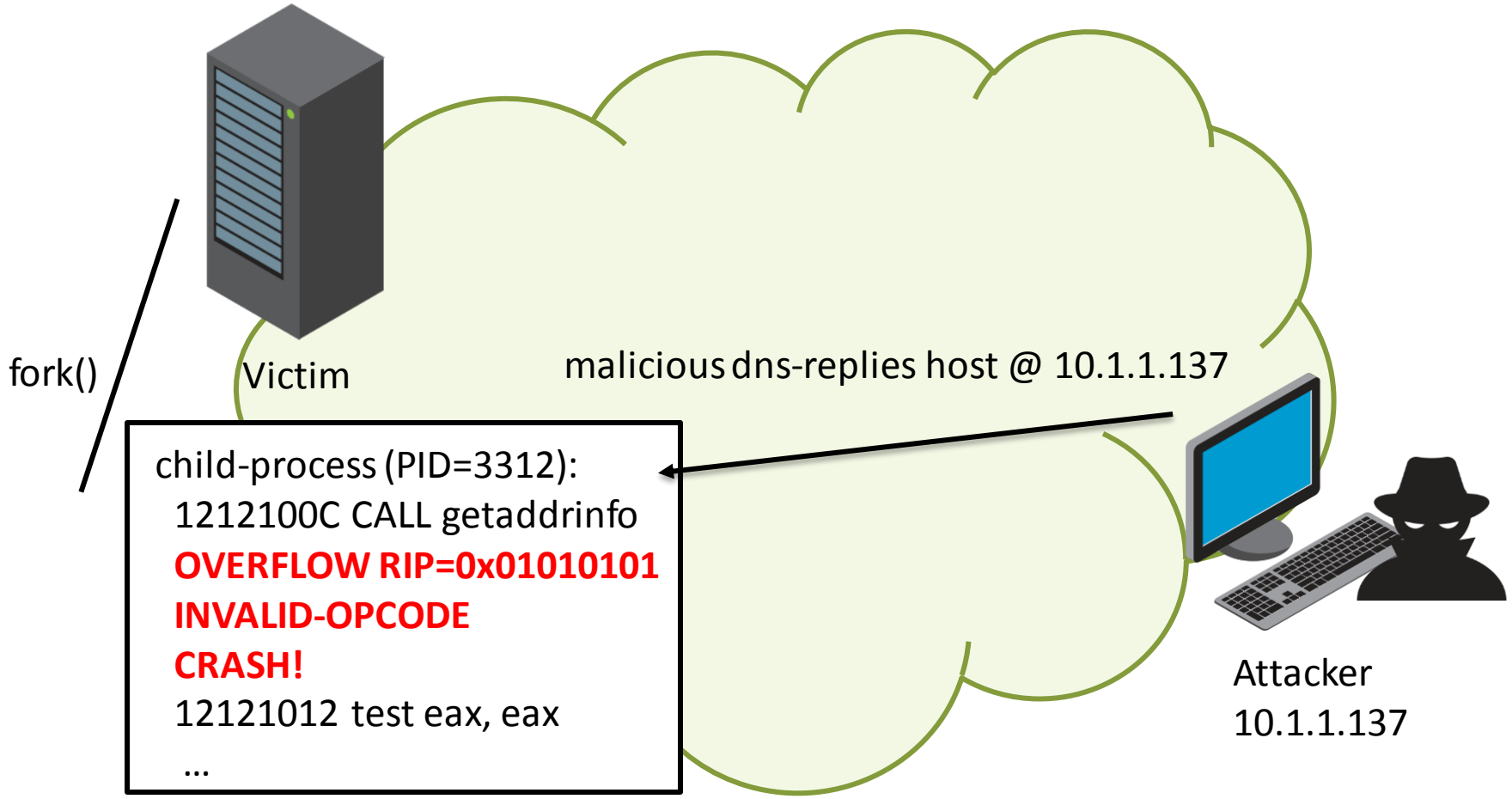
Reply Arbitrary DNS

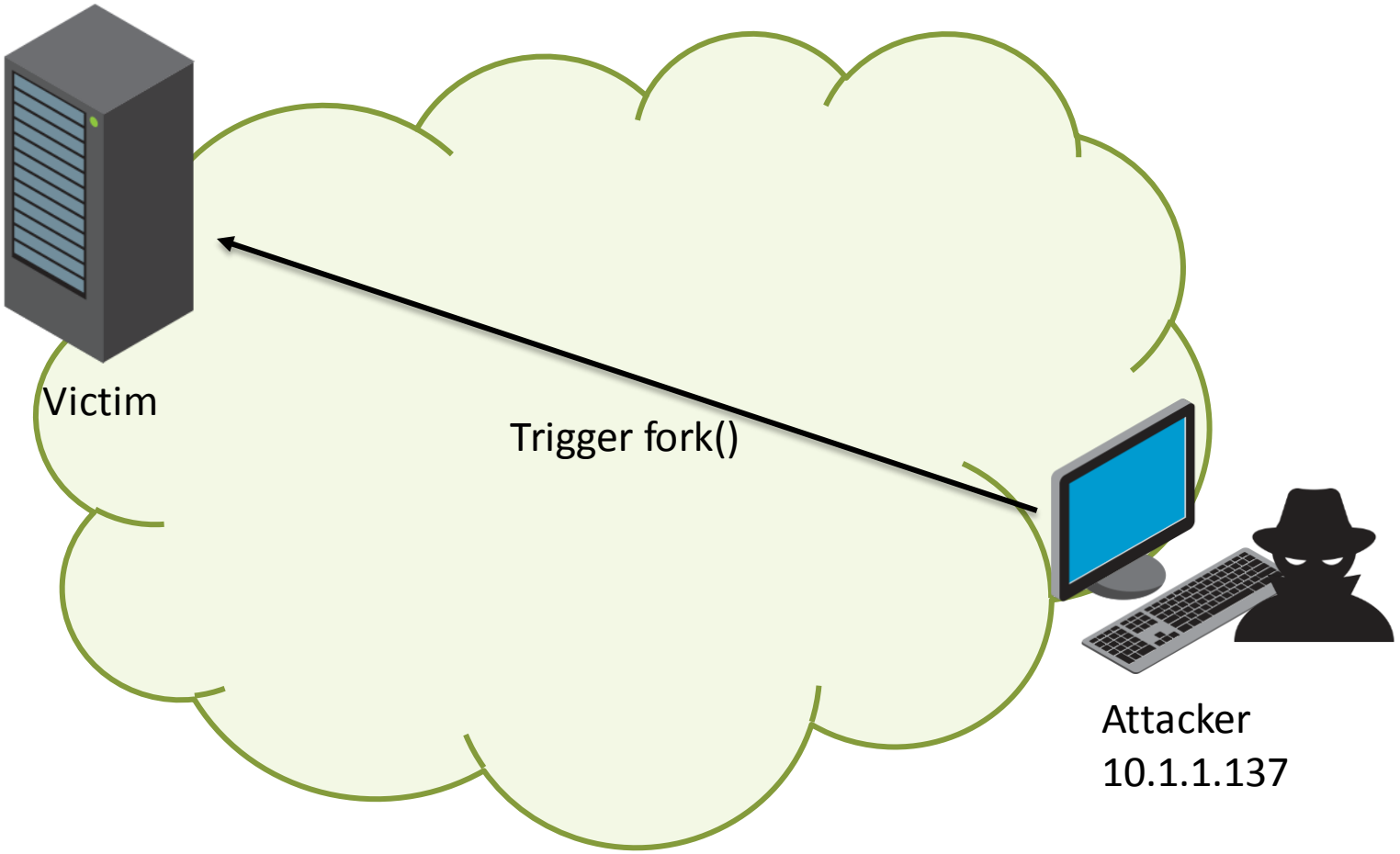
- Assume attacker can answer arbitrary DNS requests
- Acquiring the domain
- Local Arp Poisoning
- Any other way ..

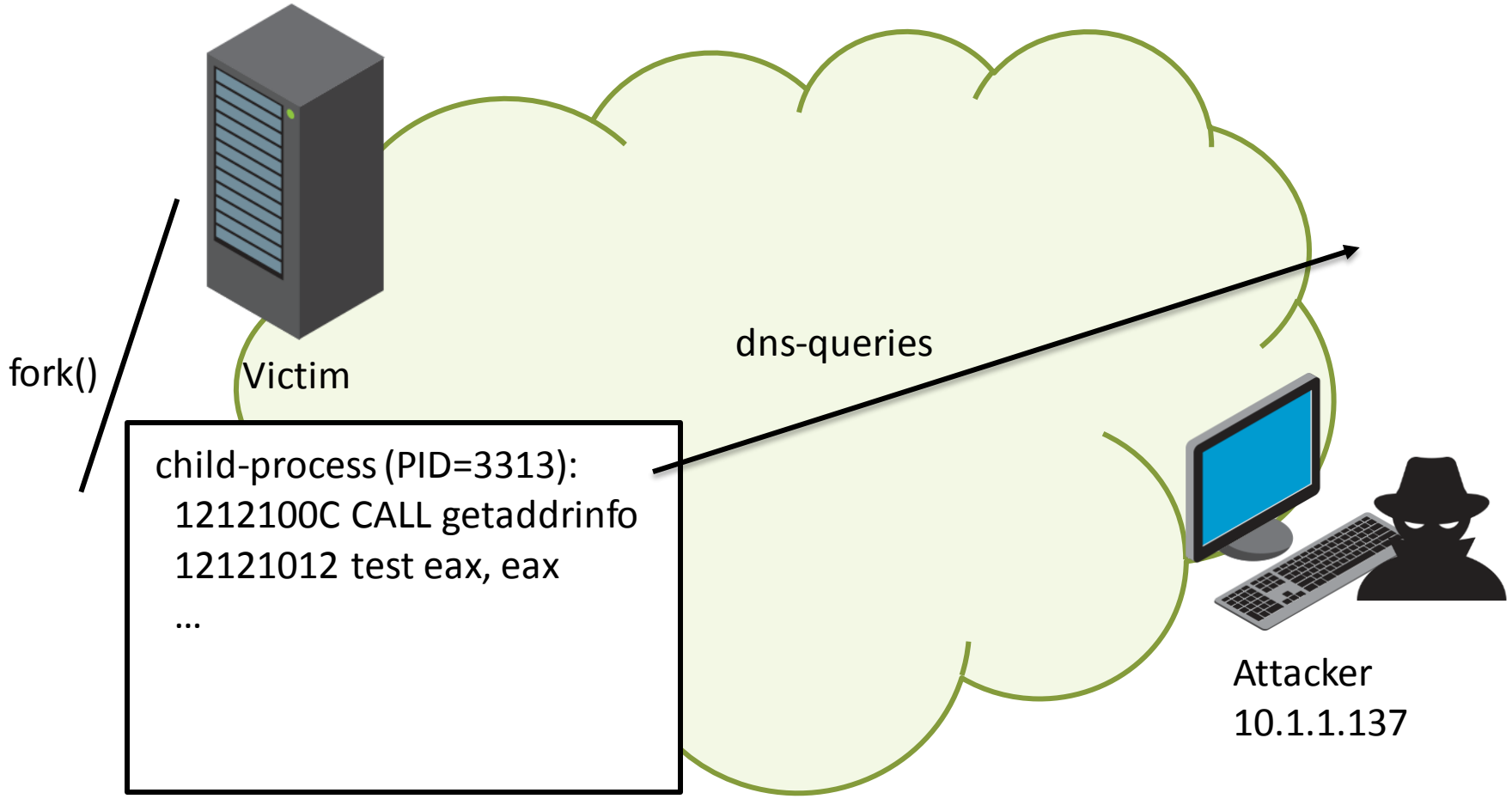

Victim
Trigger fork()
Attacker
10.1.1.137

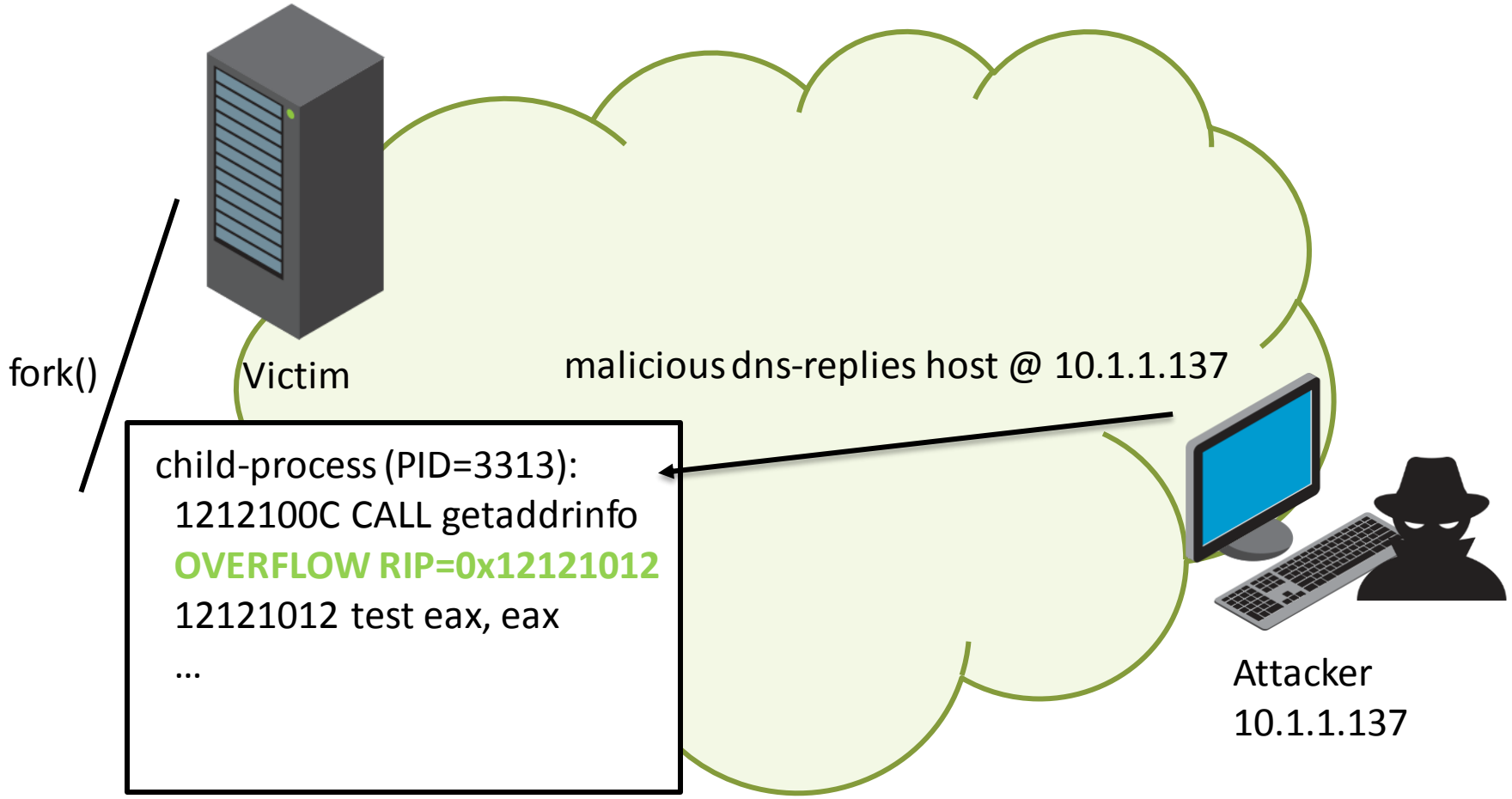

fork()
Victim
child-process (PID=3312):
Attacker
10.1.1.137

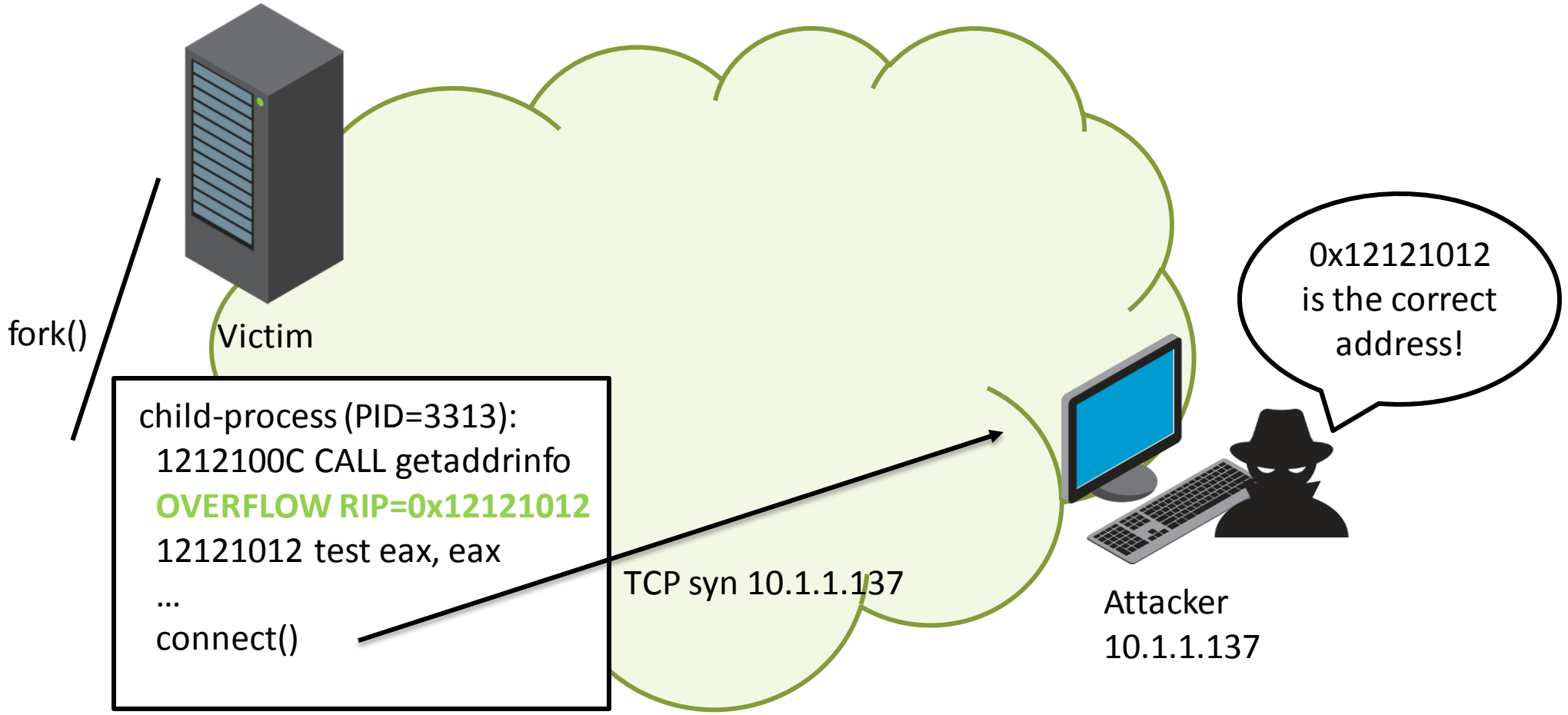

fork()
Victim
dns-queries
child-process (PID=3312):
1212100C CALL getaddrinfo
12121012 test eax, eax
…
Attacker
10.1.1.137


fork()
Victim
malicious dns-replies host @ 10.1.1.137
child-process (PID=3312):
1212100C CALL getaddrinfo
OVERFLOW RIP=0x01010101
INVALID-OPCODE
CRASH!
12121012 test eax, eax
…
Attacker
10.1.1.137


Victim
Trigger fork()
Attacker
10.1.1.137


fork()
Victim
dns-queries
child-process (PID=3313):
1212100C CALL getaddrinfo
12121012 test eax, eax
…
Attacker
10.1.1.137


fork()
Victim
malicious dns-replies host @ 10.1.1.137
child-process (PID=3313):
1212100C CALL getaddrinfo
OVERFLOW RIP=0x12121012
12121012 test eax, eax
…
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
- ~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

DNS request-response (malformed)

AAAAA..AAA../x00

Attacker

Original RIP	12	10	12	12	00	00	00	00
Overwritten RIP	00	10	12	12	00	00	00	00
	LSB							MSB

Byte by Byte Approach

Return Address	Sent Buffer	Response?
0x0000000012121000	AAA…0x00	NO
0x0000000012121001	AAA…0x01	NO
0x000000012121….	…	NO
0x0000000012121012	AAA…0x12	YES
0x0000000012120012	AAA…0x12 0x00	NO
0x0000000012120112	AAA…0x12 0x01	NO
0x000000001212….12	…	NO
0x0000000012121012	AAA…0x12 0x10	YES
….	…	

8 * 256 ☺

Finding Potentially Exploitable Applications

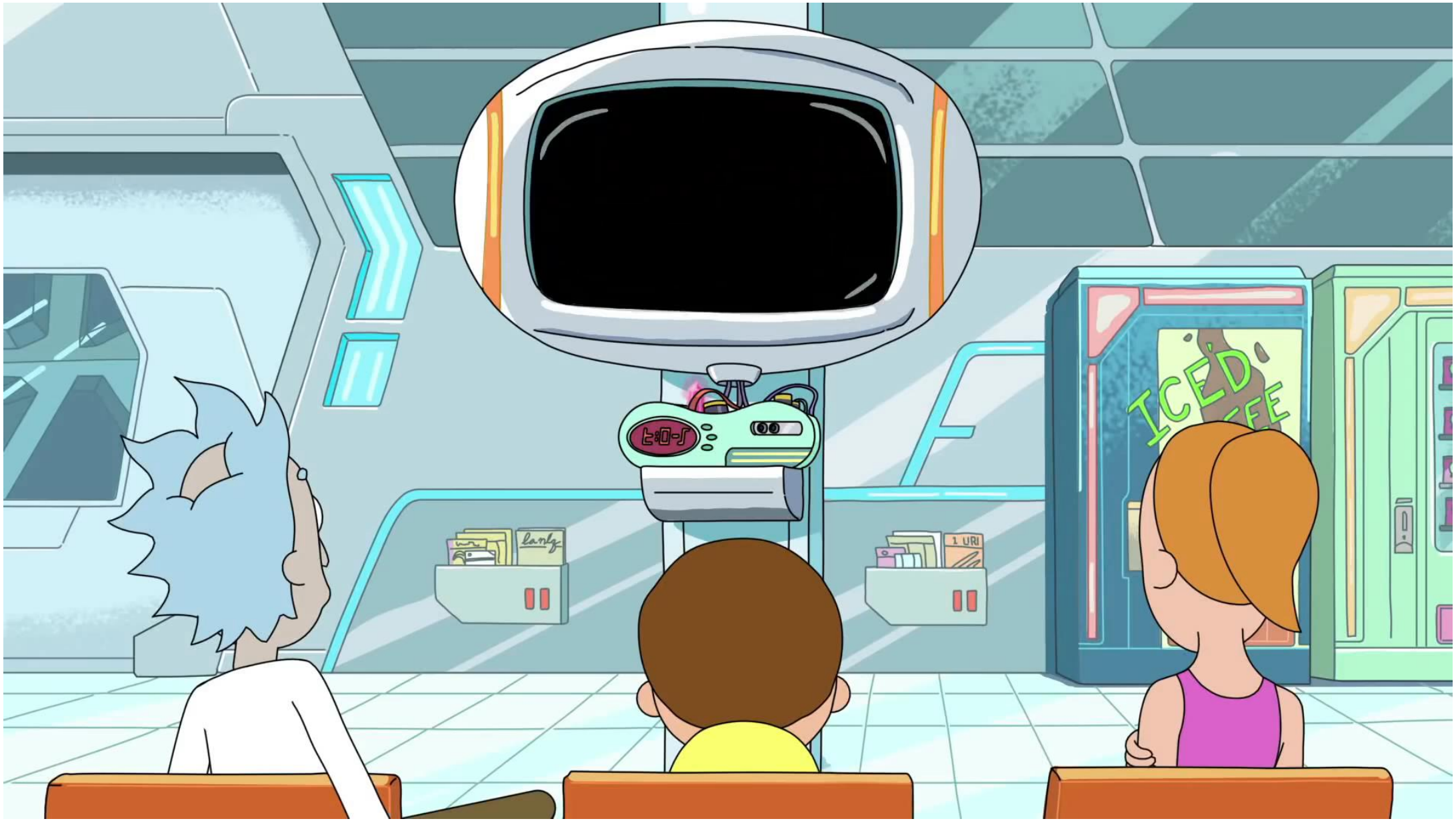
- fork() && getaddrinfo()
  - Using the correct flow
- http://codesearch.debian.net
  - Indexes source code of ~18,000 packages
- ~1,300 potential exploitable apps

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

…


ICED
1 URI

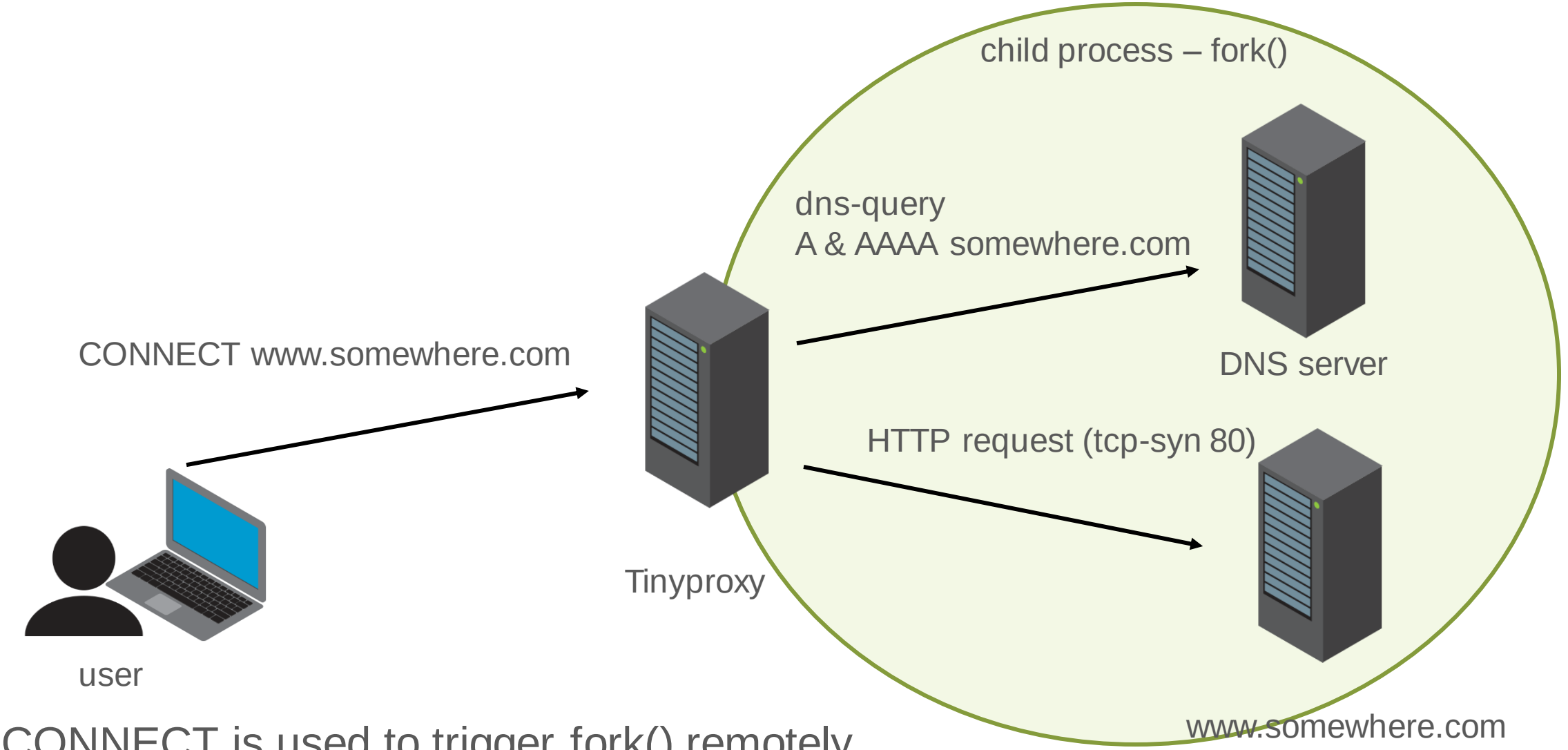
Finding Potentially Exploitable Applications

…

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

…

tinyproxy

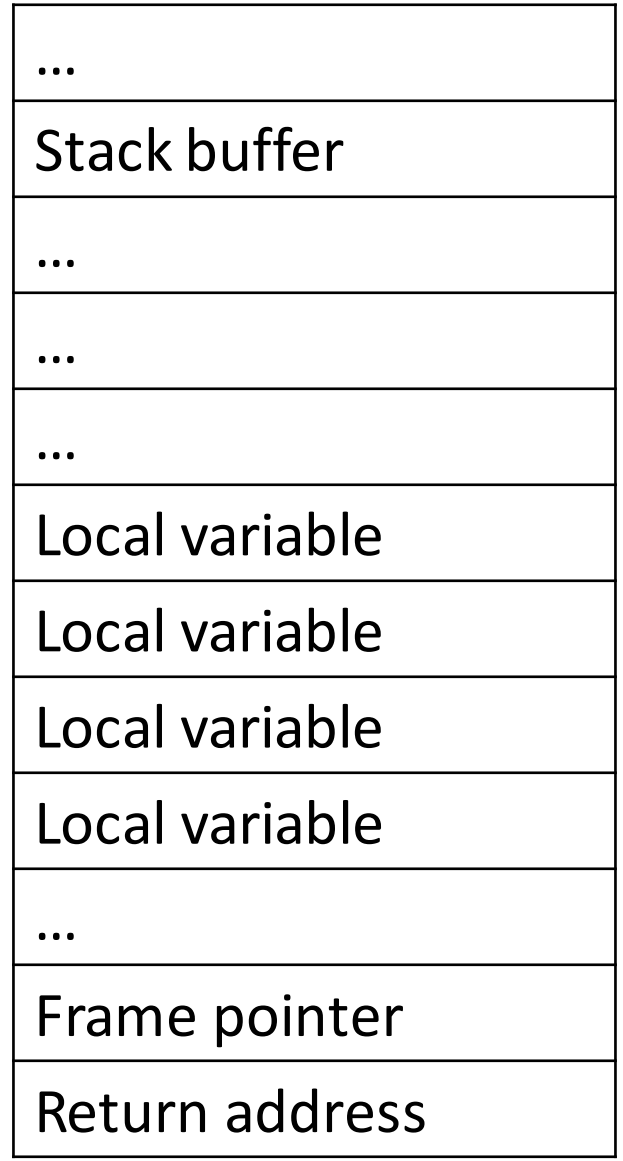


1. CONNECT is used to trigger fork() remotely
2. http-request is used as an indication of success

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 ☺



Segfault (1st)

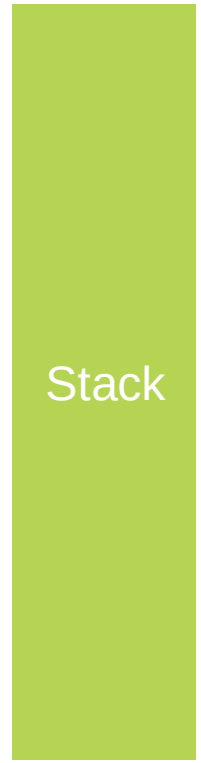
- RBX originally points the stack
- We can leak this address too!
- Address pointed by RBX *only* has to be writeable
  - Flow is not effected

```
Program received signal SIGSEGV, Segmentation fault.
0x00007f0e3d7a3b09 in ns_name_ntop () from /lib/x86_64-linux-gnu/libresolv.so.2
2: x/5i $rip
=> 0x7f0e3d7a3b09 <ns_name_ntop+441>:   mov    BYTE PTR [rbx],sil
   0x7f0e3d7a3b0c <ns_name_ntop+444>:   add    rbx,0x1
   0x7f0e3d7a3b10 <ns_name_ntop+448>:   jmp    0x7f0e3d7a3a2b <ns_name_ntop+219>
   0x7f0e3d7a3b15 <ns_name_ntop+453>:   nop    DWORD PTR [rax]
   0x7f0e3d7a3b18 <ns_name_ntop+456>:   cmp    edi,0x41
(gdb) i r rbx
rbx            0x4141414141414141       4702111234474983745
```

Leak RBX (arbitrary stack address)

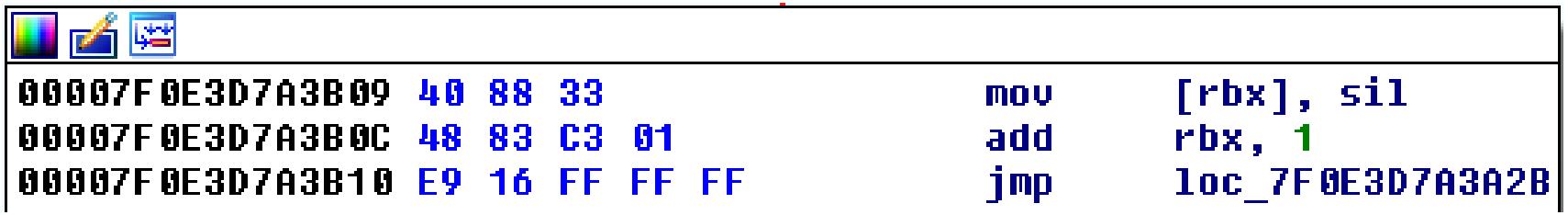
(/proc/PID/maps)

limit	base	length	
0x7ffd07882000	0x7ffd078a3000	0x21000	0x0 [stack]
0x7ffd079f0000	0x7ffd079f2000	0x2000	0x0 [vvar]



AA	A2	88	07	FD	7F	00	00

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

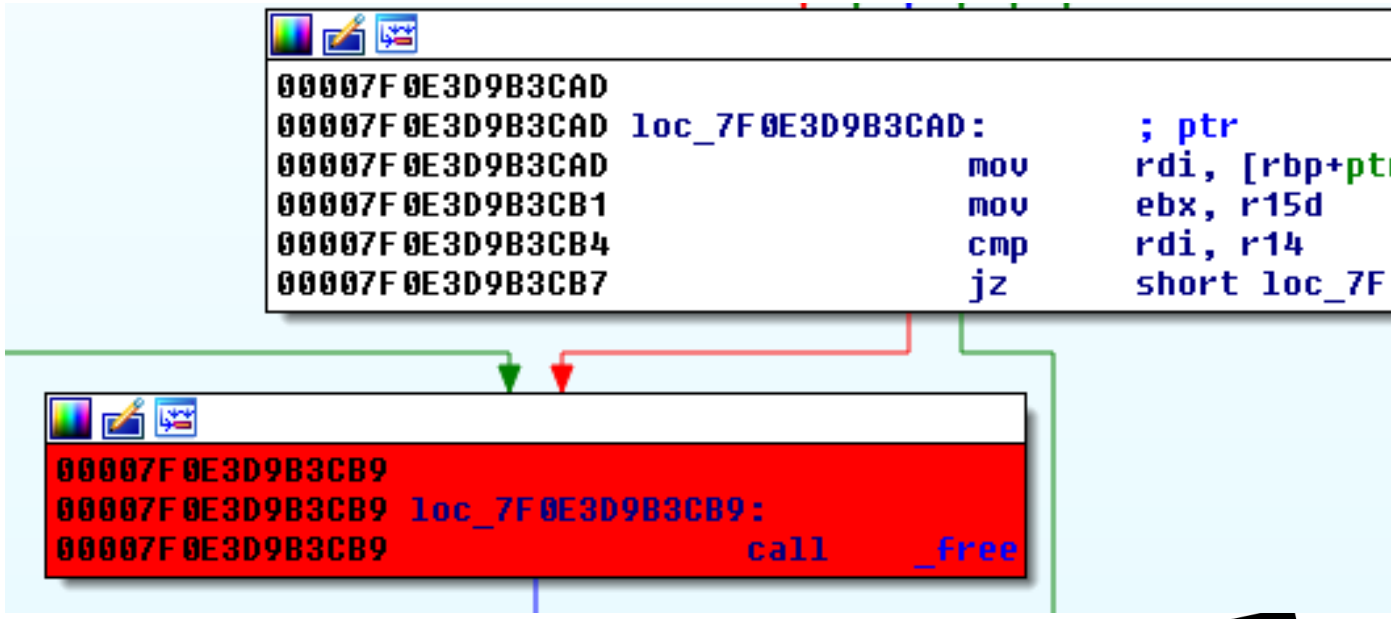


```
00007F0E3D7A3B09 40 88 33          mov     [rbx], sil
00007F0E3D7A3B0C 48 83 C3 01       add     rbx, 1
00007F0E3D7A3B10 E9 16 FF FF FF    jmp     loc_7F0E3D7A3A2B
```

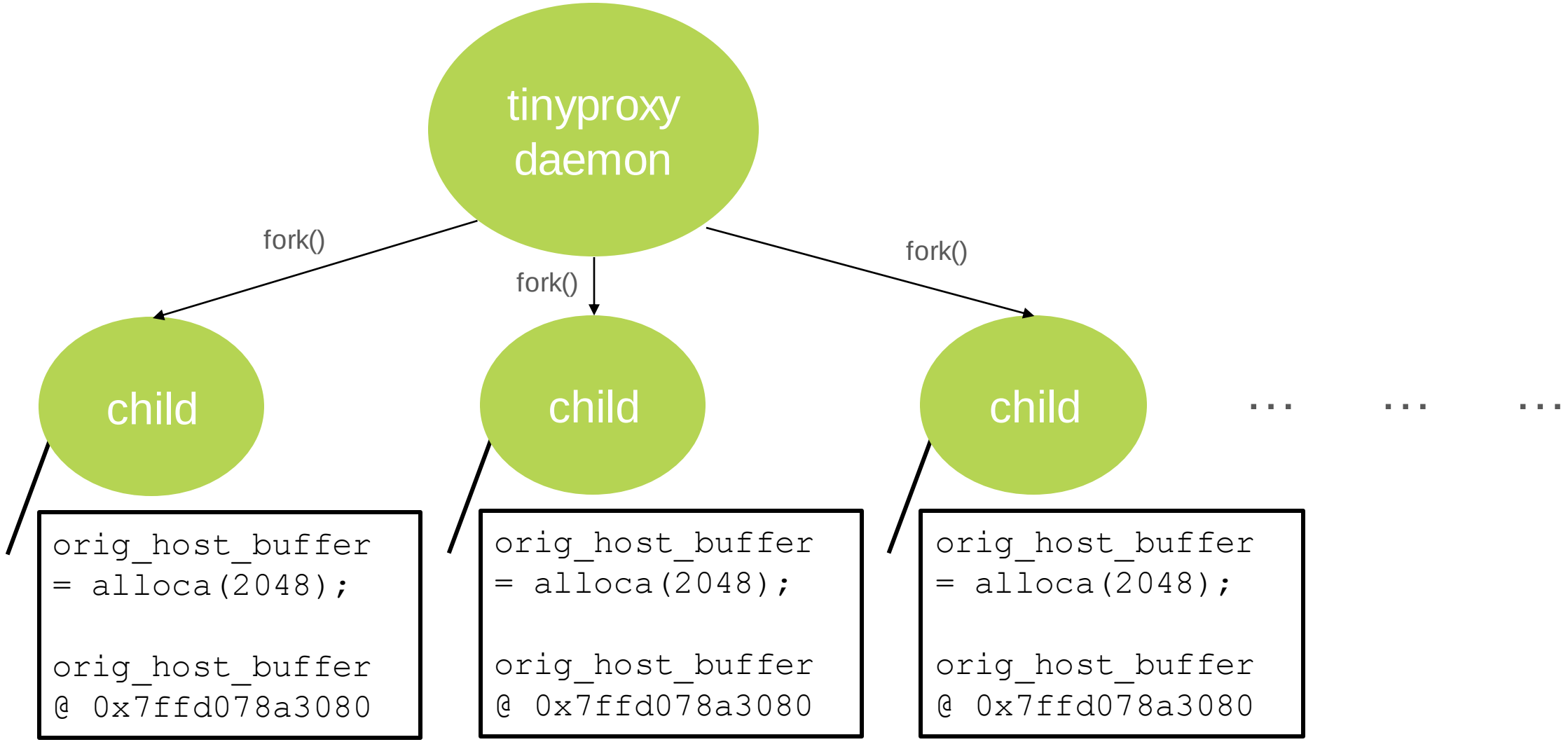
Segfault (2nd)

- 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?

```
...
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;
```



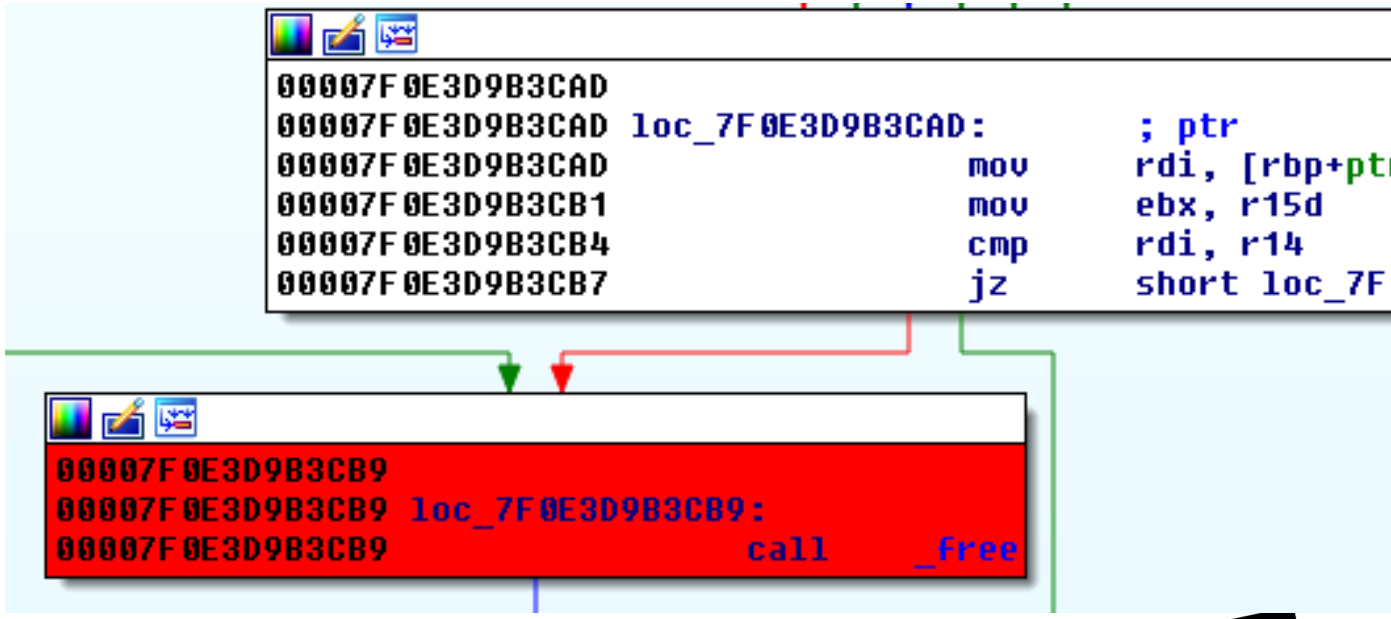
Stack is Identical for Different fork()s



Precise Pointers

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

```
...
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

rbx	*Sent Buffer*	*Response?*
0x00007fffed000000	AAA...0x00 0x00 0x00 0xed 0xff 0x7f	YES
0x00007fffed001000	AAA...0x00 0x10 0x00 0xed 0xff 0x7f	YES
0x00007fffed002000	AAA...0x00 0x20 0x00 0xed 0xff 0x7f	YES
0x00007fffed003000	AAA...0x00 0x30 0x00 0xed 0xff 0x7f	YES
0x00007fffed004000	AAA...0x00 0x40 0x00 0xed 0xff 0x7f	YES
0x00007fffed005000	AAA...0x00 0x50 0x00 0xed 0xff 0x7f	YES
0x00007fffed006000	AAA...0x00 0x60 0x00 0xed 0xff 0x7f	YES
0x00007fffed007000	AAA...0x00 0x70 0x00 0xed 0xff 0x7f	YES
0x00007fffed008000	AAA...0x00 0x80 0x00 0xed 0xff 0x7f	NO

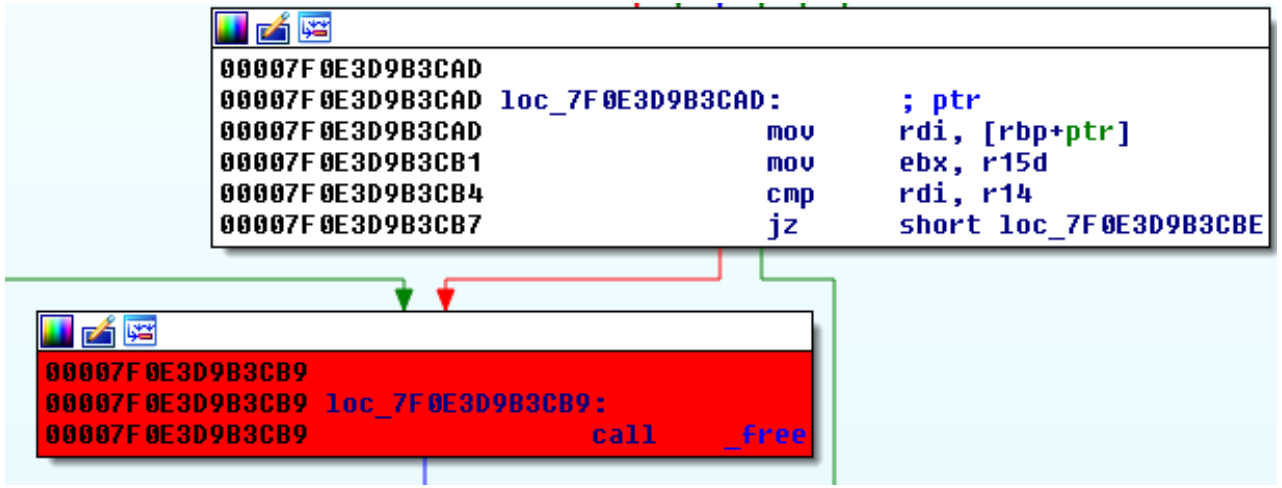
Game Over?

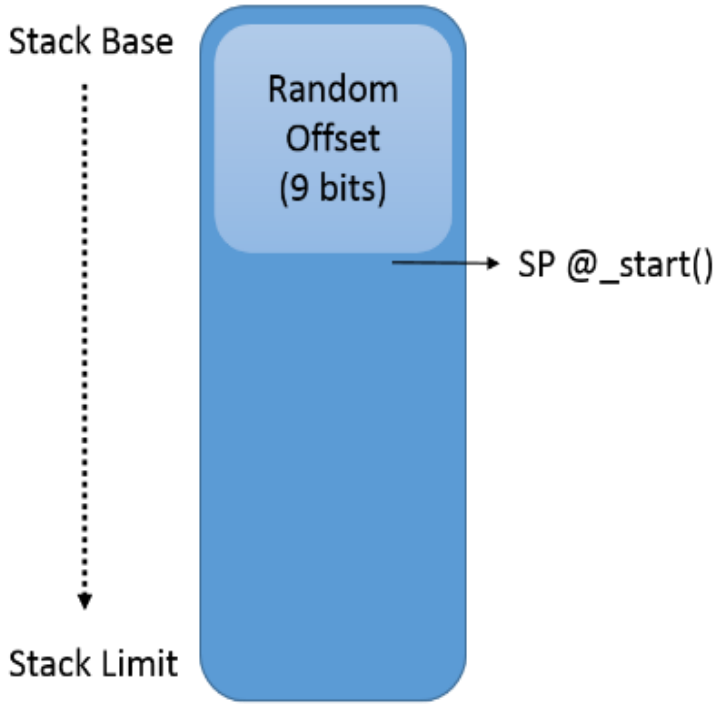
Leaked =>	0xaa0000	Stack base
	…	…
0x400 is constant offset =>	0xa9fc00 (-0x400)	orig_host_buffer
	…	…
	…	…
	…	…
	0xa9f500 (-0xb00)	Local variable
	0xa9f4f8 (-0xb08)	Local variable
		…
	0xa9f400 (-0xc00)	Frame pointer
	0xa9f3f8 (-0xc08)	Return address

Offset from Stack-Base is Constant?

/arch/x86/kernel/process.c

```
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;
}
```



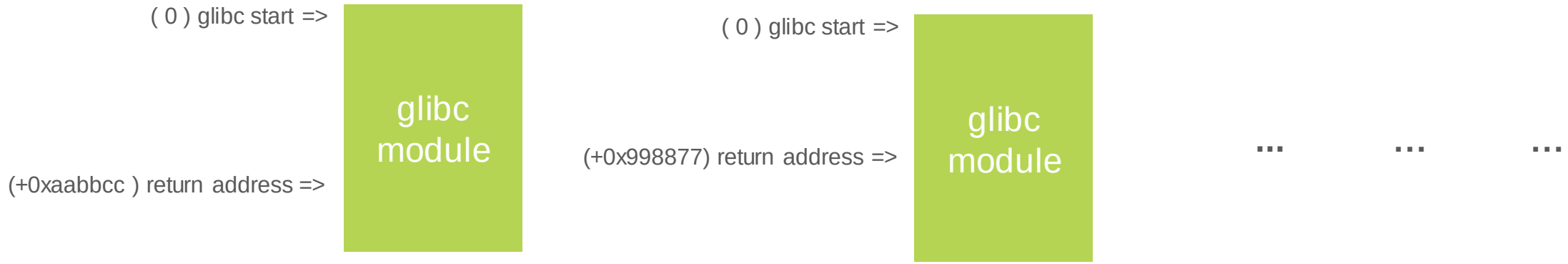


Leak libc base (for fun & gadgets)

- 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?