Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh

Stanford University
Hacking 101

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 no Crash?
Enough to build exploit

GET /blabla HTTP/1.0

HTTP/1.0 404 Not Found

GET /AAAAAAAAAAAAAAAAAAAAAAA

connection closed
Don’t even need to know what application is running!

Exploit scenarios:

1. Open source
2. Open binary
3. Closed-binary (and source)
## Attack effectiveness

- Works on 64-bit Linux with ASLR, NX and canaries

<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</td>
<td>1,950</td>
<td>5</td>
</tr>
<tr>
<td>(unknown binary and source)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
Background Outline

• Stack Vulnerabilities
• No-eXecute memory (NX)
• Return-Oriented Programming (ROP)
• Address Space Layout Randomization (ASLR)
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()
Stack vulnerabilities

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

Stack:
- return address: 0x400000
- Memory Stack:
  - AAAAAAAAAAAAAA
  - AAAAAAAAAAAAAA
  - AAAAAAAAAAAA
  - AAAAAAAA
  - AAAAA

handle_client()
Stack vulnerabilities

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

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

    return;
}
```

Stack:

```
return address
0x41414141
AAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAA
```

??
Stack vulnerabilities

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

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

    return;
}
```

Stack:
- Return address: 0x500000
- Shellcode:
  - `dup2(sock, 0);`
  - `dup2(sock, 1);`
  - `execve("/bin/sh", 0, 0);`
Stack vulnerabilities

```c
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 (NX)

2. Randomize memory addresses (ASLR)

Stack:
return address
0x600000
0x1029827189
123781923719
823719287319
879181823828

Shellcode:
dup2(sock, 0);
dup2(sock, 1);
execve("/bin/sh", 0, 0);
Non-eXecutable (NX) memory

- Stack
- Heap
- .text (code)

Read
Write
Execute

Read
Execute

Read
Write

Read
Execute

No Execute permission on .text (code)
Non-eXecutable (NX) memory

stack

heap

.text (code)

Read
Write
Execute

Read
Execute

stack

heap

.text (code)

Read
Write

Read
Execute
Return-Oriented Programming (ROP)

code fragment
...
...
...
...

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

Stack:
0x800000
Return-Oriented Programming (ROP)

.text:
  code fragment
  ...
  ...
  ...
  ...

  dup2(sock, 0);
  return;
  0x800000

  dup2(sock, 1);
  return;
  0x600000

  execve("/bin/sh", 0, 0);
  return;
  0x700000

Stack:
  0x700000
  0x600000
  0x800000

ROP gadget
Address Space Layout Randomization (ASLR)

.text: 0x400000

code fragment

dup2(sock, 0);
return;

dup2(sock, 1);
return;

execve("/bin/sh", 0, 0);
return;

Stack:

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

.text: 0x400000 + ??

code fragment

... ...

dup2(sock, 0);
return;

dup2(sock, 1);
return;

execve("/bin/sh", 0, 0);
return;

Stack:

0x700000 + ??
0x600000 + ??
0x800000 + ??
Exploit requirements today

1. Break ASLR.

2. Copy of binary (find ROP gadgets / break NX).
   - Is it even possible to hack unknown applications?
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.

```
buf[1024]  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.

Return address

00000000000000000000000000000000  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.

Return address

00000000000000000000000000000000

(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.

Return address

\[
\begin{array}{c|c}
00000000000000000000000000000000 & 0x011183 \\
\end{array}
\]
(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.
How to find gadgets?

.text:

0x401183
code fragment

0x401170 ??
0x401160 ??
0x401150 ??
0x401140 ??
0x401130 ??

Stack:
return address 0x401183
buf[1024]
How to find gadgets?

.text:

0x401183

code fragment

0x401170

crash

0x401160

??

0x401150

??

0x401140

??

0x401130

??

Connection closes

Stack:

return address
0x401170

AAAAAAAAAAAA

AAAAAAAAAAAA
How to find gadgets?

.text:

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

Connection closes

Stack:

return address 0x401160
AAAAAAAAAAAA
AAAAAAAAAAAA
How to find gadgets?

.text:

- code
- crash
- crash
- no crash
- ??
- ??

Connection hangs

Stack:

- return address: 0x401150
- AAAAAAAAAAAAAAAAAAAAAAAAA
How to find gadgets?

.text:

- code fragment
- crash
- crash
- no crash
- crash
- crash

Connection closes

Stack:
- return address
  - 0x401130
  - AAAAAAAAAA
Three types of gadgets

Stop gadget

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

• Never crashes

Crash gadget

```c
abort();
return;
```

• Always crashes

Useful gadget

```c
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;

Stack:

return address 0x401170
other

return address 0x401170
other

buf[1024]

sleep(10);
return;

0x401150

0x401170

Crash!!
Finding useful gadgets

dup2(sock, 0);
return;

0x401170

stack:

0x401150
return address
0x401170

buf[1024]

sleep(10);
return;

0x401150

No crash
How to find gadgets?

.text:

0x401183
- code
- fragment

0x401170
- crash

0x401160
- crash

0x401150
- stop gadget

0x401140
- crash

0x401130
- crash

Stack:
- other
- return address 0x401183
- buf[1024]
How to find gadgets?

.text:

0x401183

code fragment

0x401170

gadget!

0x401160

crash

0x401150

stop gadget

0x401140

crash

0x401130

crash

Connection hangs

Stack:

0x401150

return address

0x401170

AAAAAAAAAAAAAAA

AAAAAAAAAAAAAA
How to find gadgets?

.text:

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

Stack:
- 0x401150
- return address 0x401160
- AAAAAAAAAAA
- AAAAAAAAAAA

Connection closes
What are we looking for?

write(int sock, void *buf, int len)
What are we 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]

sock
rdi

buf
rsi

len
rdx

0x400000
ret addr

0x500000
buf

0x600000
len

0x700000
sock

0x700000
sock

0x700000
sock

0x700000
sock

0x700000
sock

0x700000
sock
Pieces of the puzzle

- stop gadget [call sleep]
- pop rsi
- ret
- pop rdi
- ret
- pop rdx
- ret
- call write
- ret
Pieces of the puzzle

stop gadget
[call sleep]

pop rsi
ret

pop rdi
ret

call strcmp
ret

call write
ret
Pieces of the puzzle

pop rsi
ret

pop rdi
ret

stop gadget
[call sleep]
call strcmp
ret
call write
ret
Pieces of the puzzle

The BROP gadget

pop rdi
ret

pop rsi
pop r15
ret

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

stop gadget
[call sleep]

call strcmp
ret

call write
ret

pop rdi
ret
Pieces of the puzzle

The BROP gadget

pop rdx
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
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 closes

Finding the BROP gadget

Stack:

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

Connection hangs
Finding the BROP gadget

Stack:

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

Connection hangs
Finding the BROP gadget

Stack:

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

BROP gadget:

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

Connection hangs
Attack so far

- Can control first two arguments to calls (using BROP gadget).
- Need to control third argument (using call strcmp).
- Need to find call write.
Procedure Linking Table (PLT)

Origin: 0x400000

PLT

.text
Procedure Linking Table (PLT)

Origin: 0x400000

.text
  call write
  ...
  call write
  ...
  call strcmp

PLT
  jmp [sleep]
  push 0
  jmp dlresolve

  jmp [write]
  push 1
  jmp dlresolve

  jmp [strcmp]
  push 2
  jmp dlresolve

[GOT dereference]

libc
  sleep() {
    ...
  }
  write() {
    ...
  }
  strcmp() {
Finding the PLT

- Toward beginning of .text
- Each entry is 0x10 bytes apart, 0x10 aligned
- Most don’t crash: syscalls - returnEFAULT on bad args.
- Can check if entry + 6 succeeds too (dlresolve slow path)
Fingerprinting strcmp

- Use return address (.text) as readable

<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

- `strcmp(origin, origin)` // sets RDX / write length to 7 - ELF header.
- Set rdi to socket number, rsi to origin/addr
- Try calling candidate PLT function.
- Check if data received on socket.
- Chain writes with different FD numbers to find socket. Use multiple connections.
Launching a shell

1. dump binary to network - not blind anymore!
2. dump symbol table to find PLT calls.
3. redirect stdin/out to socket:
   - dup2(fd, 0);
   - close(0); fcntl(sock, F_DUPFD, 0);
4. read() “/bin/sh” from socket to writable
   - writable can be environ (from symbol table)
5. execve(“/bin/sh”, 0, 0)
Braille

- Fully automated: from crash to shell.
- 2,000 lines of Ruby.
- Needs function that will trigger overflow:
  - nginx: 68 lines.
  - mysql: 121 lines.
  - proprietary service: 35 lines.

```
try_exp(data) → true crash
false no crash
```
Attack complexity

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

# of requests for nginx
BROP conclusions

• It's scary what a motivated attacker can do: e.g., hack unknown binaries.

• Hackers can get past point solutions - need principled solutions:
  • OS: rerandomize ASLR & canaries per-fork
  • Compiler: Encrypt or MAC pointers?
  • Application: Dune (privilege separation)?