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 no Crash?
Enough to build exploit

GET /blabla HTTP/1.0
HTTP/1.0 404 Not Found
GET /AAAAAAAAAAAAAAAAAAAAAAAA
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 (unknown binary and source)</td>
<td>1,950</td>
<td>5</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.
Outline

• Introduction.
• Background on exploits.
• Blind ROP (BROP).
• Optimizations.
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
- buf[1024]

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

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

    return;
}

handle_client()

Stack:

return address

0x400000

AAAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAAA
AAAAAAAAAAAAA
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;
}

return address
0x500000

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;
}

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)

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

text:

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

Stack:
0x600000
0x102982
71891237
81923719
82371928
73198791
81823828

Executable
Non-Executable

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

code fragment

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

Stack:
0x800000
Return-Oriented Programming (ROP)

.text:

code fragment

.. ...

dup2(sock, 0);
return;

dup2(sock, 1);
return;

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

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.

Return address

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

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
00000000000000000000000000000000
(Was: 0x401183)
```

```
0x401183
```
How to find gadgets?

.text:

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

Stack:

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

Stack:
- return address: 0x401170
- AAAAAAAAAAA
- AAAAAAAAAAAA

Connection closes
How to find gadgets?

.text:

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

Connection closes

Stack:

return address
0x401160
AAAAAAAAAAAA
AAAAAAAAAAA
How to find gadgets?

.text:

- 0x401183: code fragment
- 0x401170: crash
- 0x401160: crash
- 0x401150: no crash
- 0x401140: ??
- 0x401130: ??

Connection hangs

Stack:
- return address 0x401150
- AAAAAAAAAAAA
How to find gadgets?

.text:

0x401183
- code
- fragment

0x401170
- crash

0x401160
- crash

0x401150
- no crash

0x401140
- crash

0x401130
- crash

Connection closes

Stack:
- return address 0x401130
- AAAAAAAAAAA
- AAAAAAAAAAA
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;

return address 0x401170
buf[1024]

Stack:
other
return address 0x401170

sleep(10);
return;

0x401170
Crash!!

0x401150
Finding useful gadgets

dup2(sock, 0);
return;

Stack:

return address
0x401170

buf[1024]

sleep(10);
return;

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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401183</td>
<td>gadet!</td>
</tr>
<tr>
<td>0x401170</td>
<td></td>
</tr>
<tr>
<td>0x401160</td>
<td>crash</td>
</tr>
<tr>
<td>0x401150</td>
<td>stop gadget</td>
</tr>
<tr>
<td>0x401140</td>
<td>crash</td>
</tr>
<tr>
<td>0x401130</td>
<td>crash</td>
</tr>
</tbody>
</table>

Connection hangs

Stack:

- Return address: 0x401170
- AAAA
- AAAA
- AAAA
- AAAA
- AAAA
- AAAA
- AAAA
- AAAA
- AAAA
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
AAAAAAAAAAA
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]
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]

pop rbx
ret

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
Pieces of the puzzle

The BROP gadget

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

- call write
- 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]
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

The PLT

- stop gadget
  [call sleep]
- call strcmp
- ret
- call write
- ret
Procedure Linking Table (PLT)

---

**.text:**
- PLT
  - call write
  - call strcmp
  - ...

**PLT**
- jmp [sleep]
- jmp [write]
- jmp [strcmp]
- jmp [dup2]
- jmp [execve]
- jmp [...]
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>
Finding write

- Try sending data to socket by 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 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)
Braille

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

\[
\text{try\_exp(data)} \rightarrow \begin{cases} 
\text{true} & \text{crash} \\
\text{false} & \text{no crash}
\end{cases}
\]
Attack complexity

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

# of requests for nginx