Outline

Format string attacks
Return address protections
Announcements intermission
ASLR and counterattacks
W+X (DEP)
Epilogue: BCLPR Makefile

Format string attack

- Attacker-controlled format: little interpreter
- Step one: add extra integer specifiers, dump stack
  - Already useful for information disclosure

Format string attack layout

Format string attack: overwrite

- %n specifier: store number of chars written so far to pointer arg
- Advance format arg pointer to other attacker-controlled data
- Control number of chars written with padding
- On x86, use unaligned stores to create pointer
Outline

Format string attacks
Return address protections
Announcements intermission
ASLR and counterattacks
W_X (DEP)
Epilogue: BCLPR Makefile

Canary in the coal mine

Adjacent Canary Idea

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00</td>
</tr>
<tr>
<td>1</td>
<td>0x0D</td>
</tr>
<tr>
<td>2</td>
<td>0xA</td>
</tr>
</tbody>
</table>

Terminator Canary

- Value hard to reproduce because it would tell the copy to stop
- StackGuard: 0x00 0D 0A FF
- String functions:
  - fgets(), etc.
- getc()
- Carriage return: similar to newline?
- Doesn't stop: memcpy, custom loops

Random Canary

- Can't reproduce because attacker can't guess
- For efficiency, usually one per execution
- Ineffective if disclosed

XOR Canary

- Want to protect against non-sequential overwrites
- XOR return address with value c at entry
- XOR again with c before return
- Standard choice for c: see random canary
Further refinements

- More flexible to do earlier in compiler
- Rearrange buffers after other variables
  - Reduce chance of non-control overwrite
- Skip canaries for functions with only small variables
  - Who has an overflow bug in an 8-byte array?

What’s usually not protected?

- Backwards overflows
- Function pointers
- Adjacent structure fields
- Adjacent static data objects

Where to keep canary value

- Fast to access
- Buggy code/attacker can’t read or write
- Linux/x86: \%gs: 0x14

Complex anti-canary attack

- Canary not updated on fork in server
- Attacker controls number of bytes overwritten

Complex anti-canary attack

- Canary not updated on fork in server
- Attacker controls number of bytes overwritten
- ANRY BNRY CNRY DNRY ENRY FNRY
- search $2^{32} \rightarrow$ search $4 \cdot 2^8$

Shadow return stack

- Suppose you have a safe place to store the canary
- Why not just store the return address there?
- Needs to be a separate stack
- Ultimate return address protection
You may notice

- We’re catching up with the readings
- Today: StackGuard, ASLR attacks
- Next time: CFI, Shacham ROP

Pre-proposals due tomorrow

- Most groups formed?
- One PDF per group, include schedule choices
- Submit via Moodle by 11:55pm

BCLPR vulnerability discovered!

- Back door fixed in version 1.1
  - Also added -v option
- Upgrade your own VM, source on website
- test-exploit also updated

HA1 week 1 feedback

- Future attacks will be harder
  - At least 7 vulnerabilities still exist
- Double-check submission logistics

Outline

- Format string attacks
- Return address protections
- Announcements intermission
- ASLR and counterattacks
- W⊕X (DEP)
- Epilogue: BCLPR Makefile
Basic idea

- “Address Space Layout Randomization”
- Move memory areas around randomly so attackers can’t predict addresses
- Keep internal structure unchanged
  - E.g., whole stack moves together

Code and data locations

- Execution of code depends on memory location
- E.g., on 32-bit x86:
  - Direct jumps are relative
  - Function pointers are absolute
  - Data must be absolute

Relocation (Windows)

- Extension of technique already used in compilation
- Keep table of absolute addresses, instructions on how to update
- Disadvantage: code modifications take time on load, prevent sharing

PIC/PIE (GNU/Linux)

- “Position-Independent Code / Executable”
- Keep code unchanged, use register to point to data area
- Disadvantage: code complexity, register pressure hurt performance

What’s not covered

- Main executable (Linux 32-bit PIC)
- Incompatible DLLs (Windows)
- Relative locations within a module/area

Entropy limitations

- Intuitively, entropy measures amount of randomness, in bits
- Random 32-bit int: 32 bits of entropy
- ASLR page aligned, so at most $32 - 12 = 20$ bits of entropy
- Other constraints further reduce possibilities
Leakage limitations

- If an attacker learns the randomized base address, can reconstruct other locations
- Any stack address $\rightarrow$ stack unprotected, etc.

GOT hijack (Müller)

- Main program fixed, libc randomized
- PLT in main program used to call libc
- Rewrite PLT to call attacker’s favorite libc functions
  - E.g., turn printf into system

GOT hijack (Müller)

printf@plt: jmp *0x8049678
...
system@plt: jmp *0x804967c
...
0x8049678: <addr of printf in libc>
0x804967c: <addr of system in libc>

ret2pop (Müller)

- Take advantage of shellcode pointer already present on stack
- Rewrite intervening stack to treat the shellcode pointer like a return address
  - A long sequence of chained returns, one pop

ret2pop (Müller)

Outline

Format string attacks
Return address protections
Announcements intermission
ASLR and counterattacks
$W \oplus X$ (DEP)
Epilogue: BCLPR Makefile
Basic idea

- Traditional shellcode must go in a memory area that is writable, so the shellcode can be inserted executable, so the shellcode can be executed.
- But benign code usually does not need this combination.
- $W \ XOR X$, really: $(W \land X)$

Non-writable code, $X \rightarrow \neg W$

- E.g., read-only .text section
- Has been standard for a while, especially on Unix
- Lets OS efficiently share code with multiple program instances

Non-executable data, $W \rightarrow \neg X$

- Prohibit execution of static data, stack, heap.
- Not a problem for most programs.
  - Incompatible with some GCC features no one uses.
  - Non-executable stack opt-in on Linux, but now near-universal.

Implementing $W \oplus X$

- Page protection implemented by CPU.
- Some architectures (e.g. SPARC) long supported $W \oplus X$.
- x86 historically did not.
  - One bit controls both read and execute.
  - Partial stop-gap "code segment limit".
- Eventual obvious solution: add new bit.
  - NX (AMD), XD (Intel), XN (ARM).

One important exception

- Remaining important use of self-modifying code: just-in-time (JIT) compilers.
  - E.g., all modern JavaScript engines.
- Allow code to re-enable execution per-block.
  - mprotect, VirtualProtect.
  - Now a favorite target of attackers.

Counterattack: code reuse

- Attacker can’t execute new code.
- So, take advantage of instructions already in binary.
- There are usually a lot of them.
- And no need to obey original structure.
Classic return-to-libc (1997)

- Overwrite stack with copies of:
  - Pointer to libc's `system` function
  - Pointer to "/bin/sh" string (also in libc)
- The `system` function is especially convenient
- Distinctive feature: return to entry point

Chained return-to-libc

- Shellcode often wants a sequence of actions, e.g.
  - Restore privileges
  - Allow execution of memory area
  - Overwrite system file, etc.
- Can put multiple fake frames on the stack
  - Basic idea present in 1997, further refinements

Beyond return-to-libc

- Can we do more? Oh, yes.
- Classic academic approach: what's the most we could ask for?
- Here: “Turing completeness”
- How to do it: reading for Thursday

Outline

- Format string attacks
- Return address protections
- Announcements intermission
- ASLR and counterattacks
- W(X (DEP)
- Epilogue: BCLPR Makefile

BCLPR Makefile

```
CFLAGS := -g -w -m32 \ 
  -fno-stack-protector \ 
  -z execstack -z norelro
```

Standard non-security options
BCLPR Makefile

CFLAGS := -g -w -m32 \
  -fno-stack-protector \
  -z execstack -z norelro

- Turn off canaries
- Allow execution on stack
- Leave GOT writable

BCLPR unprotection, cont’d

- Not in Makefile: disable ASLR
- Is done system-wide in VM
- For non-VM testing, can use
  setarch i386 -R

Next time

- Return-oriented programming (ROP)
  - And counter-defenses
- Control-flow integrity (CFI)