

CSci 5271  
Introduction to Computer Security  
Day 7: Defensive programming and design,  
part 1

Stephen McCamant  
University of Minnesota, Computer Science & Engineering

## Outline

- Return-oriented programming (ROP)
- Control-flow integrity (CFI)
- More modern exploit techniques
- Announcements intermission
- Saltzer & Schroeder's principles
- More secure design principles
- Software engineering for security
- Secure use of the OS

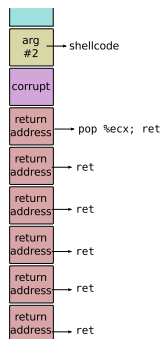
## Basic new idea

- Treat the stack like a new instruction set
- "Opcodes" are pointers to existing code
- Generalizes return-to-libc with more programmability

## 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

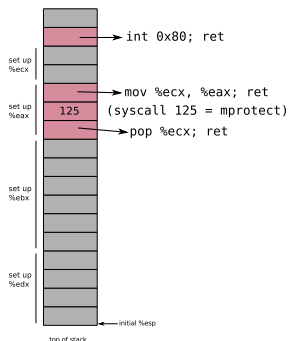
## ret2pop (Müller)



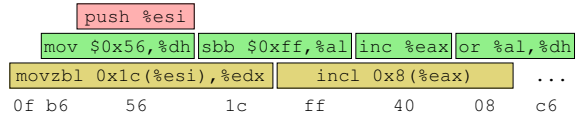
## Gadgets

- Basic code unit in ROP
- Any existing instruction sequence that ends in a return
- Found by (possibly automated) search

## Another partial example



## Overlapping x86 instructions



- Variable length instructions can start at any byte
- Usually only one intended stream

## Where gadgets come from

- Possibilities:
  - Entirely intended instructions
  - Entirely unaligned bytes
  - Fall through from unaligned to intended
- Standard x86 return is only one byte, 0xc3

## Building instructions

- String together gadgets into manageable units of functionality
- Examples:
  - Loads and stores
  - Arithmetic
  - Unconditional jumps
- Must work around limitations of available gadgets

## Hardest case: conditional branch

- Existing jCC instructions not useful
- But carry flag CF is
- Three steps:
  - Do operation that sets CF
  - Transfer CF to general-purpose register
  - Add variable amount to %esp

## Further advances in ROP

- Can also use other indirect jumps, overlapping not required
- Automation in gadget finding and compilers
- In practice: minimal ROP code to allow transfer to other shellcode

## Counterfeit OO Prog. (S&P'15)

- Idea: construct fake objects with pointers to real C++ vtables

```
for (i = 0; i < nStudents; i++)
    students[i]->dropCourse(id);
```

- Overlapping object fields facilitate data flow

## Anti-ROP: lightweight

- Check stack sanity in critical functions
- Check hardware-maintained log of recent indirect jumps (kBouncer)
- Unfortunately, exploitable gaps

## Gaps in lightweight anti-ROP

- Three papers presented at 2014's USENIX Security
- Hide / flush jump history
- Very long loop → context switch
- Long "non-gadget" fragment
- (Later: call-preceded gadgets)

## Anti-ROP: still research

- Modify binary to break gadgets
- Fine-grained code randomization
- Beware of adaptive attackers ("JIT-ROP")
- Next up: control-flow integrity

## Outline

Return-oriented programming (ROP)

Control-flow integrity (CFI)

More modern exploit techniques

Announcements intermission

Saltzer & Schroeder's principles

More secure design principles

Software engineering for security

Secure use of the OS

## Basic CFI principle

- Each indirect jump should only go to a programmer-intended (or compiler-intended) target
- I.e., enforce call graph
- Often: identify disjoint target sets

## Approximating the call graph

- One set: all legal indirect targets
- Two sets: indirect calls and return points
- n sets: needs possibly-difficult points-to analysis

## Target checking: classic

- Identifier is a unique 32-bit value
- Can embed in effectively-nop instruction
- Check value at target before jump
- Optionally add shadow stack

## Target checking: classic

```
cmp [ecx], 12345678h
jne error_label
lea ecx, [ecx+4]
jmp ecx
```

## Challenge 1: performance

- In CCS'05 paper: 16% avg., 45% max.
  - Widely varying by program
  - Probably too much for on-by-default
- Improved in later research
  - Common alternative: use tables of legal targets

## Challenge 2: compatibility

- Compilation information required
- Must transform entire program together
- Can't inter-operate with untransformed code

## Recent advances: COTS

- Commercial off-the-shelf binaries
- CCFIR (Berkeley+PKU, Oakland'13): Windows
  - Use Windows ASLR information to find targets
- CFI for COTS Binaries (Stony Brook, USENIX'13): Linux
  - Keep copy of original binary, build translation table

## Control-Flow Guard

- CFI-style defense now in latest Windows systems
- Compiler generates tables of legal targets
- At runtime, table managed by kernel, read-only to user-space

## Coarse-grained counter-attack

- "Out of Control" paper, Oakland'14
- Limit to gadgets allowed by coarse policy
  - Indirect call to function entry
  - Return to point after call site ("call-preceded")
- Use existing direct calls to VirtualProtect
- Also used against kBouncer

## Code-pointer Integrity (CPI, OSDI'14)

- Memory safety would block attacks, but expensive
  - Spatial safety: e.g., no buffer overflows
  - Temporal safety: e.g., no use after free
- Idea: apply memory safety only to objects containing code pointers
  - Similar benefits to good CFI
- Challenge: low-overhead memory isolation

## Control-flow bending counter-attack

- Control-flow attacks that still respect the CFG
- Especially easy without a shadow stack
- Printf-oriented programming generalizes format-string attacks

## Outline

Return-oriented programming (ROP)  
Control-flow integrity (CFI)  
More modern exploit techniques  
Announcements intermission  
Saltzer & Schroeder's principles  
More secure design principles  
Software engineering for security  
Secure use of the OS

## Target #1: web browsers

- Widely used on desktop and mobile platforms
- Easily exposed to malicious code
- JavaScript is useful for constructing fancy attacks

## Heap spraying

- How to take advantage of uncontrolled jump?
- Maximize proportion of memory that is a target
- Generalize NOP sled idea, using benign allocator
- Under  $W \oplus X$ , can't be code directly

## JIT spraying

- Can we use a JIT compiler to make our sleds?
- Exploit unaligned execution:
  - Benign but weird high-level code (bitwise ops. with constants)
  - Benign but predictable JITted code
  - Becomes sled + exploit when entered unaligned

## JIT spray example

```
25 90 90 90 3c and $0x3c909090,%eax
25 90 90 90 3c and $0x3c909090,%eax
25 90 90 90 3c and $0x3c909090,%eax
25 90 90 90 3c and $0x3c909090,%eax
```

## JIT spray example

```
90          nop
90          nop
90          nop
3c 25      cmp $0x25,%a1
90          nop
90          nop
90          nop
3c 25      cmp $0x25,%a1
```

## Use-after-free

- Low-level memory error of choice in web browsers
- Not as easily audited as buffer overflows
- Can lurk in attacker-controlled corner cases
- JavaScript and Document Object Model (DOM)

## Sandboxes and escape

- Chrome NaCl: run untrusted native code with SFI
  - Extra instruction-level checks somewhat like CFI
- Each web page rendered in own, less-trusted process
- But not easy to make sandboxes secure
  - While allowing functionality

## Chained bugs in Pwnium 1

- Google-run contest for complete Chrome exploits
  - First edition in spring 2012
- Winner 1: 6 vulnerabilities
- Winner 2: 14 bugs and “missed hardening opportunities”
- Each got \$60k, bugs promptly fixed

## Outline

Return-oriented programming (ROP)  
Control-flow integrity (CFI)  
More modern exploit techniques  
Announcements intermission  
Saltzer & Schroeder’s principles  
More secure design principles  
Software engineering for security  
Secure use of the OS

## Research with Prof. Kangjie Lu

- Prof. Kangjie Lu is interested in security, program analysis, and operating systems
- Starting a project on Linux kernel bugs
- For more details see forum post

## Supplemental office hours tomorrow

- Tomorrow (Thursday), 11am-noon in 4-225E

## Alternative Saltzer & Schroeder

- Not a replacement for reading the real thing, but:
  - <http://emergentchaos.com/the-security-principles-of-saltzer-and-schroeder>
- Security Principles of Saltzer and Schroeder, illustrated with scenes from Star Wars (Adam Shostack)

## Deadlines reminder

- Exercise set 1: Thursday night
- HA1 week 3: Friday night

## Outline

Return-oriented programming (ROP)  
Control-flow integrity (CFI)  
More modern exploit techniques  
Announcements intermission  
Saltzer & Schroeder's principles  
More secure design principles  
Software engineering for security  
Secure use of the OS

## Economy of mechanism

- Security mechanisms should be as simple as possible
- Good for all software, but security software needs special scrutiny

## Fail-safe defaults

- When in doubt, don't give permission
- Whitelist, don't blacklist
- Obvious reason: if you must fail, fail safe
- More subtle reason: incentives

## Complete mediation

- Every mode of access must be checked
  - Not just regular accesses: startup, maintenance, etc.
- Checks cannot be bypassed
  - E.g., web app must validate on server, not just client

## Open design

- Security must not depend on the design being secret
- If anything is secret, a minimal key
  - Design is hard to keep secret anyway
  - Key must be easily changeable if revealed
  - Design cannot be easily changed

## Open design: strong version

- "The design should not be secret"
- If the design is fixed, keeping it secret can't help attackers
- But an unscrutinized design is less likely to be secure



## Separation of privilege

- ▣ Real world: two-person principle
- ▣ Direct implementation: separation of duty
- ▣ Multiple mechanisms can help if they are both required
  - Password and `wheel` group in Unix

## Least privilege

- ▣ Programs and users should have the most limited set of powers needed to do their job
- ▣ Presupposes that privileges are suitably divisible
  - Contrast: Unix `root`

## Least privilege: privilege separation

- ▣ Programs must also be divisible to avoid excess privilege
- ▣ Classic example: multi-process OpenSSH server
- ▣ N.B.: Separation of privilege  $\neq$  privilege separation

## Least common mechanism

- ▣ Minimize the code that all users must depend on for security
- ▣ Related term: minimize the Trusted Computing Base (TCB)
- ▣ E.g.: prefer library to system call; microkernel OS

## Psychological acceptability

- ▣ A system must be easy to use, if users are to apply it correctly
- ▣ Make the system's model similar to the user's mental model to minimize mistakes

## Sometimes: work factor

- ▣ Cost of circumvention should match attacker and resource protected
- ▣ E.g., length of password
- ▣ But, many attacks are easy when you know the bug

## Sometimes: compromise recording

- Recording a security failure can be almost as good as preventing it
- But, few things in software can't be erased by `root`

## Outline

- Return-oriented programming (ROP)
- Control-flow integrity (CFI)
- More modern exploit techniques
- Announcements intermission
- Saltzer & Schroeder's principles
- More secure design principles
- Software engineering for security
- Secure use of the OS

## Pop quiz

- What's the type of the return value of `getchar`?
- Why?

## Separate the control plane

- Keep metadata and code separate from untrusted data
- Bad: format string vulnerability
- Bad: old telephone systems

## Defense in depth

- Multiple levels of protection can be better than one
- Especially if none is perfect
- But, many weak security mechanisms don't add up

## Canonicalize names

- Use unique representations of objects
- E.g. in paths, remove `.`, `..`, extra slashes, symlinks
- E.g., use IP address instead of DNS name

## Fail-safe / fail-stop

- If something goes wrong, behave in a way that's safe
- Often better to stop execution than continue in corrupted state
- E.g., better segfault than code injection

## Outline

Return-oriented programming (ROP)  
Control-flow integrity (CFI)  
More modern exploit techniques  
Announcements intermission  
Saltzer & Schroeder's principles  
More secure design principles  
Software engineering for security  
Secure use of the OS

## Modularity

- Divide software into pieces with well-defined functionality
- Isolate security-critical code
  - Minimize TCB, facilitate privilege separation
  - Improve auditability

## Minimize interfaces

- Hallmark of good modularity: clean interface
- Particularly difficult:
  - Safely implementing an interface for malicious users
  - Safely using an interface with a malicious implementation

## Appropriate paranoia

- Many security problems come down to missing checks
- But, it isn't possible to check everything continuously
- How do you know when to check what?

## Invariant

- A fact about the state of a program that should always be maintained
- Assumed in one place to guarantee in another
- Compare: proof by induction

## Pre- and postconditions

- Invariants before and after execution of a function
- Precondition: should be true before call
- Postcondition: should be true after return

## Dividing responsibility

- Program must ensure nothing unsafe happens
- Pre- and postconditions help divide that responsibility without gaps

## When to check

- At least once before any unsafe operation
- If the check is fast
- If you know what to do when the check fails
- If you don't trust
  - your caller to obey a precondition
  - your callee to satisfy a postcondition
  - yourself to maintain an invariant

## Sometimes you can't check

- Check that  $p$  points to a null-terminated string
- Check that  $fp$  is a valid function pointer
- Check that  $x$  was not chosen by an attacker

## Error handling

- Every error must be handled
  - i.e, program must take an appropriate response action
- Errors can indicate bugs, precondition violations, or situations in the environment

## Error codes

- Commonly, return value indicates error if any
- Bad: may overlap with regular result
- Bad: goes away if ignored

## Exceptions

- Separate from data, triggers jump to handler
- Good: avoid need for manual copying, not dropped
- May support: automatic cleanup (finally)
- Bad: non-local control flow can be surprising

## Testing and security

- “Testing shows the presence, not the absence of bugs” – Dijkstra
- Easy versions of some bugs can be found by targeted tests:
  - Buffer overflows: long strings
  - Integer overflows: large numbers
  - Format string vulnerabilities: %x

## Fuzz testing

- Random testing can also sometimes reveal bugs
- Original ‘fuzz’ (Miller): program `</dev/urandom`
- Modern: small random changes to a benign input

## Outline

Return-oriented programming (ROP)  
Control-flow integrity (CFI)  
More modern exploit techniques  
Announcements intermission  
Saltzer & Schroeder’s principles  
More secure design principles  
Software engineering for security  
Secure use of the OS

## Avoid special privileges

- Require users to have appropriate permissions
  - Rather than putting trust in programs
- Anti-pattern 1: setuid/setgid program
- Anti-pattern 2: privileged daemon
- But, sometimes unavoidable (e.g., email)

## One slide on setuid/setgid

- Unix users and process have a user id number (UID) as well as one or more group IDs
- Normally, process has the IDs of the user who starts it
- A setuid program instead takes the UID of the program binary

## Don't use shells or Tcl

- ... in security-sensitive applications
- String interpretation and re-parsing are very hard to do safely
- Eternal Unix code bug: path names with spaces

## Prefer file descriptors

- Maintain references to files by keeping them open and using file descriptors, rather than by name
- References same contents despite file system changes
- Use `openat`, etc., variants to use FD instead of directory paths

## Prefer absolute paths

- Use full paths (starting with `/`) for programs and files
- `$PATH` under local user control
- Initial working directory under local user control
  - But FD-like, so can be used in place of `openat` if missing

## Prefer fully trusted paths

- Each directory component in a path must be write protected
- Read-only file in read-only directory can be changed if a parent directory is modified

## Don't separate check from use

- Avoid pattern of e.g., `access` then `open`
- Instead, just handle failure of `open`
  - You have to do this anyway
- Multiple references allow races
  - And `access` also has a history of bugs

## Be careful with temporary files

- Create files exclusively with tight permissions and never reopen them
  - See detailed recommendations in Wheeler
- Not quite good enough: reopen and check matching device and inode
  - Fails with sufficiently patient attack

## Give up privileges

- ▣ Using appropriate combinations of `set*id` functions
  - Alas, details differ between Unix variants
- ▣ Best: give up permanently
- ▣ Second best: give up temporarily
- ▣ Detailed recommendations: Setuid Demystified (USENIX'02)

## Whitelist environment variables

- ▣ Can change the behavior of called program in unexpected ways
- ▣ Decide which ones are necessary
  - As few as possible
- ▣ Save these, remove any others

## Next time

- ▣ Recommendations from the author of `qmail`
- ▣ A variety of isolation mechanisms