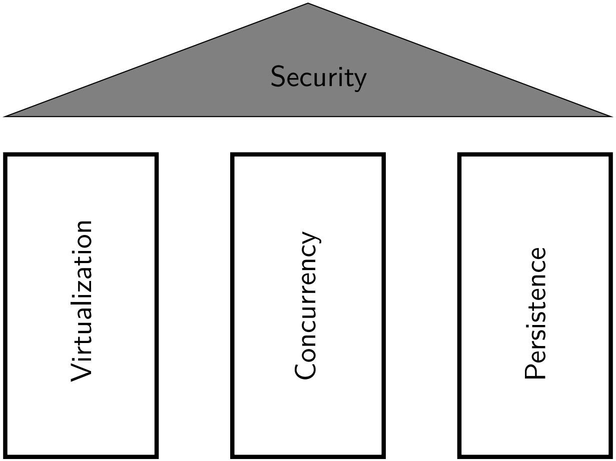
Security

CSC 343, Operating Systems

Security



Topics covered in this lecture

- Software testing
- Fuzzing
- Sanitization
- Data Execution Prevention
- Address Space Layout Randomization
- Stack canaries
- Control-Flow Integrity (CFI)

This slide deck covers chapters 5.3 and 6.4 in SS3P.

Why testing?

Testing is the process of ***executing code*** *to* ***find errors****.*

An error is a deviation between observed behavior and specified behavior, i.e., a violation of the underlying specification:

- Functional requirements (features a, b, c)
- Operational requirements (performance, usability)
- Security requirements?

Limitations of testing

Testing can only show the presence of bugs, never their absence. (Edsger W. Dijkstra)

A successful test finds a deviation. Testing is a form of dynamic analysis. Code is executed, the testing environment observes the behavior of the code, detecting violations.

- Key advantage: reproducible, generally testing gives you the concrete input for failed test cases.
- Key disadvantage: complete testing of all control-flow/data-flow paths reduces to the halting problem, in practice, testing is hindered due to state explosion.

Forms of testing

- Manual testing
- Fuzz testing
- Symbolic and concolic testing

We focus on *security* testing or testing to find *security* bugs, that is, bugs that are reachable through attacker-controlled inputs.

Recommended reading: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World

Coverage as completeness metric

Intuition: A software flaw is only detected if the flawed statement is executed. Effectiveness of test suite therefore depends on how many statements are executed.

How to measure code coverage?

Several approaches exist, all rely on instrumentation:

- gcov: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
- SanitizerCoverage: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

Sampling may reduce collection cost at slight loss of precision.

Fuzzing

Fuzz testing (fuzzing) is an automated software testing technique. Key idea: execute the target program with an input and check if it crashes. The fuzzing engine automatically generates new inputs based on some criteria:

- Random mutation
- Leveraging input structure
- Leveraging program structure

The inputs are then run on the test program and, if it crashes, a crash report is generated.

Fuzzing effectiveness

- Fuzzing finds bugs effectively (CVEs—unique bug numbers)
- Proactive defense during software development/testing
- Preparing offense, as part of exploit development

Fuzz input generation

Fuzzers generate new input based on generations or mutations.

- ***Generation-based*** input generation produces new input seeds in each round, independent from each other.
- ***Mutation-based*** input generation leverages existing inputs and modifies them based on feedback from previous rounds.

Fuzz input structure awareness

Programs accept some form of *input/output*. Generally, the input/output is structured and follows some form of protocol.

- ***Dumb fuzzing*** is unaware of the underlying structure.
- ***Smart fuzzing*** is aware of the protocol and modifies the input accordingly.

Example: a checksum at the end of the input. A dumb fuzzer will likely fail the checksum.

Fuzz program structure awareness

Input is processed by the program, based on the *program structure* (past executions), input can be adapted to trigger new conditions.

- ***White-box*** fuzzing leverages (expensive) semantic program analysis to mutate input; often does not scale
- ***Grey-box*** leverages program instrumentation based on previous inputs; light runtime cost, scales to large programs
- ***Black-box*** fuzzing is unaware of the program structure; often cannot explore beyond simple/early functionality

Fuzzer challenges: coverage wall

- After certain iterations the fuzzer no longer makes progress
- Hard to satisfy checks
- Chains of checks
- Leaps in input changes

Fuzzer challenges: coverage wall

Bypassing the coverage wall is hard, the following lists some approaches:

- Better input (seeds) can mitigate the coverage wall
- Fuzz individual components by writing fuzzer stubs (LibFuzzer)
- Better mutation operators (help the fuzzer guide exploration)
- Stateful fuzzing (teach fuzzer about different program states)
- Grammar-aware fuzzing (teach fuzzer about input grammar)

Fault detection

- How do we detect program faults?
- Test cases detect bugs through
  - Assertions (`assert(var != 0x23 && "var has illegal value");`) detect violations
  - Segmentation faults
  - Division by zero traps
  - Uncaught exceptions
  - Mitigations triggering termination
- How can you increase the chances of detecting a bug?

Sanitization

Sanitizers enforce a given policy, detect bugs earlier and increase effectiveness of testing. Most sanitizers rely on a combination of static analysis, instrumentation, and dynamic analysis.

- The program is analyzed during compilation (for example, to learn properties such as type graphs or to enable optimizations)
- The program is instrumented, often to record metadata at certain places and to enforce metadata checks at other places.
- At runtime, the instrumentation constantly verifies that the policy is not violated.

What policies are interesting? What metadata do you need? Where would you check?

AddressSanitizer (1/2)

AddressSanitizer (ASan) detects memory errors. It places red zones around objects and checks those objects on trigger events. ASan detects the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (configurable)
- Use-after-scope (configurable)
- Double-free, invalid free
- Memory leaks (experimental)

Typical slowdown introduced by AddressSanitizer is 2x.

AddressSanitizer (2/2)

Goal: detect memory safety violations (both spatial and temporal)

Key idea: allocate redzones (prohibited area around memory objects), check each memory access if it targets a redzone.

- What kind of metadata would you record? Where?
- What kind of operations would you instrument?
- What kind of optimizations could you think of?

ASan Metadata

- Record live objects, guard them by placing redzones around them.
- ASan uses a table that maps each 8-byte word in memory to one byte in the table. Advantage: simple address calculation (`offset+address>>3`); disadvantage: memory overhead.
- ASan stores accessibility of each word as metadata (that is, is a given address accessible or not).

ASan runtime library

- Initializes shadow map at startup
- Replaces malloc/free to update metadata (and pad allocations with redzones)
- Intercepts special functions such as `memset`

ASan policy

- Instrument every single access, check for poison value in shadow table
- Advantage: fast checks
- Disadvantage: large memory overhead (especially on 64 bit), still slow (2x)

LeakSanitizer

- LeakSanitizer detects run-time memory leaks. It can be combined with AddressSanitizer to get both memory error and leak detection, or used in a stand-alone mode.
- LSan adds almost no performance overhead until process termination, when the extra leak detection phase runs.

MemorySanitizer

- MemorySanitizer detects uninitialized reads. Memory allocations are tagged and uninitialized reads are flagged.
- Typical slowdown of MemorySanitizer is 3x.
- Note: do not confuse MemorySanitizer and AddressSanitizer.

UndefinedBehaviorSanitizer

- UndefinedBehaviorSanitizer (UBSan) detects undefined behavior. It instruments code to trap on typical undefined behavior in C/C++ programs. Detectable errors are:
    - Unsigned/misaligned pointers
    - Signed integer overflow
    - Conversion between floating point types leading to overflow
    - Illegal use of NULL pointers
    - Illegal pointer arithmetic
    - ...
- Slowdown depends on the amount and frequency of checks. This is the only sanitizer that can be used in production. For production use, a special minimal runtime library is used with minimal attack surface.

ThreadSanitizer

- ThreadSanitizer detects data races between threads. It instruments writes to global and heap variables and records which thread wrote the value last, allowing detecting of WAW, RAW, WAR data races.
- Typical slowdown is 5-15x with 5-15x memory overhead.

HexType

- HexType detects type safety violations. It records the true type of allocated objects and makes all type casts explicit.
- Typical slowdown is 0.5x.

Sanitizers

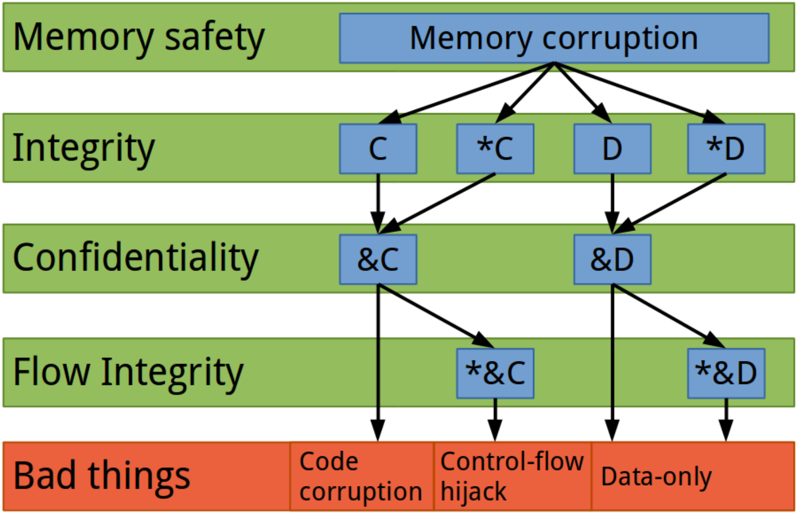
- AddressSanitizer: https://clang.llvm.org/docs/AddressSanitizer.html
- LeakSanitizer: https://clang.llvm.org/docs/LeakSanitizer.html
- MemorySanitizer: https://clang.llvm.org/docs/MemorySanitizer.html
- UndefinedBehaviorSanitizer: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
- ThreadSanitizer: https://clang.llvm.org/docs/ThreadSanitizer.html
- HexType: https://github.com/HexHive/HexType

Use sanitizers to test your code. More sanitizers are in development.

Testing Summary and conclusion

- Software testing finds bugs before an attacker can exploit them
- Manual testing: write test cases to trigger exceptions
- Fuzz testing automates and randomizes testing
- Sanitizers allow early bug detection, not just on exceptions
- AddressSanitizer is the most commonly used sanitizer and enforces probabilistic memory safety by recording metadata for every allocated object and checking every memory read/write.

Model for Control-Flow Hijack Attacks



Widely-adopted defense mechanisms

- Hundreds of defense mechanisms were proposed
- Only few mitigations were adopted
- Factors that increase chances of adoption:
    - Mitigation of the most imminent problem
    - (Very) low performance overhead
    - Fits into the development cycle

Attack vector: code injection

- Simplest form of code execution
- Generally consists of two steps:
  - Inject `code` somewhere into the process
  - Redirect control-flow to injected code

Data Execution Prevention (DEP)

- No distinction between code and data (for example, x86, ARM)
- Any data in the process could be interpreted as code (code injection: an attacker redirects control-flow to a buffer that contains attacker-controlled data as shellcode)
- ***Defense assumption:*** if an attacker cannot inject code (as data), then a code execution attack is not possible.

DEP implementation

- HW extension, add NX-bit (No eXecute) to page table entry
  - Intel calls this per-page bit XD (eXecute Disable)
  - AMD calls it Enhanced Virus Protection
  - ARM calls it XN (eXecute Never)
- This is an additional bit for every mapped virtual page. If the bit is set, then data on that page cannot be interpreted as code and the processor will trap if control flow reaches that page.

DEP summary

- DEP is now enabled widely by default (whenever a hardware support is available such as for x86 and ARM)
- Stops all code injection
- Check for DEP with checksec.sh
- DEP may be disabled through gcc flags: `-z execstack`

Attacks evolve: from code injection to reuse

- Did DEP solve all code execution attacks?
- Unfortunately not! But attacks got (much?) harder
- A code injection attack consists of two stages:
  a) redirecting control flow
  b) to injected code
- DEP prohibits execution of injected code
  - DEP does not stop the redirection of control flow
  - Attackers can still hijack control flow to ***existing*** code

Code reuse

- The attacker can overwrite a code pointer (for example, a function pointer, or a return pointer on the stack)
- Prepare the right parameters on the stack, reuse a full function (or part of a function)

From Code Reuse to full ROP

Instead of targeting a simple function, we can target a gadget

- Gadgets are a sequence of instructions ending in an indirect control-flow transfer (for example, return, indirect call, indirect jump)
- Prepare data and environment so that, for example, pop instructions load data into registers
- A gadget invocation frame consists of a sequence of 0 to n data values and an pointer to the next gadget. The gadget uses the data values and transfers control to the next gadget

Link to simple ROP tutorial

Address Space Randomization (ASR)

The security improvement of ASR depends on (i) the available entropy for randomized locations, (ii) the completeness of randomization (i.e., are all objects randomized), and (iii) the absence of information leaks.

- Successful control-flow hijack attacks depend on the attacker overwriting a code pointer with a known alternate target
- ASR changes (randomizes) the process memory layout
- If the attacker does not know where a piece of code (or data) is, then it cannot be reused in an attack
- Attacker must first ***learn*** or recover the address layout

Candidates for randomization

- Trade-off between overhead, complexity, and security benefit.
- Randomize start of heap
- Randomize start of stack
- Randomize start of code (PIE for executable, PIC each library)
- Randomize mmap allocated regions
- Randomize individual allocations (malloc)
- Randomize the code itself, e.g., gap between functions, order of functions, basic blocks, . . .
- Randomize members of structs, e.g., padding, order.

Different forms of fine-grained randomization exist. Software diversity is a related concept.

Address Space *Layout* Randomization (ASLR)

ASLR is a practical form of ASR.

- ASLR focuses on blocks of memory
- Heap, stack, code, executable, mmap regions
- ASLR is inherently page-based

ASLR entropy

- Assume start addresses of all sections are randomized
- Entropy of each section is key to security
- Attacker targets section with lowest entropy
- Early ASLR implementations had low entropy on the stack and no entropy on x86 for the executable (non-PIE executables)
- Linux (through Exec Shield) uses 19 bits of entropy for the stack (on 16 byte period) and 8 bits of mmap entropy (on 4096 byte period).

Stack canaries

- Attacks relied on a stack-based buffer overflow to inject code
- Memory safety would mitigate this problem but adding full safety checks is not feasible due to high performance overhead
- Key insight: buffer overflows require pointer arithmetic
    - Instead of checking each memory dereference during function execution, we check the integrity of a variable once
- ***Assumption:*** we only prevent ROP control-flow hijack attacks
- We therefore only need to protect the integrity of the return instruction pointer

Stack canaries

- Place a canary after a potentially vulnerable buffer
- Check the integrity of the canary before the function returns
- The compiler may place all buffers at the end of the stack frame and the canary just before the first buffer. This way, all non-buffer local variables are protected as well.
- Limitation: the stack canary only protects against ***continuous overwrites*** iff the attacker does ***not know*** the canary
- An alternative is to encrypt the return instruction pointer by xoring it with a secret

Other mitigations

- Fortify source: protect against format string attacks
- Safe exception handling: protect against popping exception frames

Control-Flow Integrity

CFI is a defense mechanism that protects applications against control-flow hijack attacks. A successful CFI mechanism ensures that the control-flow of the application never leaves the predetermined, valid control-flow that is defined at the source code/application level. This means that an attacker cannot redirect control-flow to alternate or new locations.

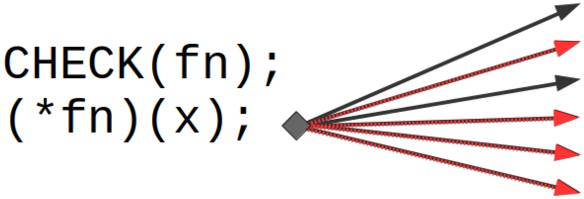


Figure 1: CFI target restriction

Basics of a CFI mechanism

Core idea: restrict the dynamic control flow of the application to the control-flow graph of the application.

- Target set construction
- Dynamic enforcement mechanism to execute runtime checks

CFI: target set construction

How do we infer the control-flow graph (for C/C++ programs)? A static analysis (on source code or binary) can recover an approximation of the control-flow graph. Precision of the analysis is crucial!

- Valid functions
- Arity
- Function prototypes
- Class hierarchy analysis

CFI: target set construction

- Trade-off between precision and compatibility.
- A single set of ***valid functions*** is highly compatible with other software but results in imprecision due to the large set size
- ***Class hierarchy analysis*** results in small sets but may be incompatible with other source code and some programmer patterns (for example, casting to void or not passing all parameters)

CFI: limitations

- CFI allows the underlying bug to fire and the memory corruption can be controlled by the attacker. The defense only detects the deviation after the fact, that is, when a corrupted pointer is used in the program
- Over-approximation in the static analysis reduces security guarantees
- Some attacks remain possible, for example, an attacker is free to modify the outcome of any conditional jump (`if` clauses depend on unprotected data value)

OS support for mitigation and sanitization

- Fault or trap signal: a segmentation fault serves as a fast and efficient way to interrupt and stop execution.
- Virtual address space: the OS controls this important abstraction and during program instantiation the OS can introduce randomness and diversity to make exploitation more costly
- Segments: the OS enables thread-local data by repurposing segment registers, stack canaries are stored in thread-local data
- Virtual address space: not all memory needs to be mapped to physical memory, enabling shadow data structures as used for sanitization
- Access to new architecture features such as Intel MPK (memory protection keys), ARM PAC (pointer authentication codes), shadow stacks, . . .

Mitigation Summary and conclusion

- Deployed mitigations do not stop all attacks
- ***Data Execution Prevention*** stops code injection attacks, but does not stop code reuse attacks
- ***Address Space Layout Randomization*** is probabilistic, shuffles memory space, prone to information leaks
- ***Stack Canaries*** are probabilistic, do not protect against direct overwrites, prone to information leaks
- ***CFI*** restricts control-flow hijack attacks, does not protect against data-only attacks