Cs 645: Lecture 3
Software Security Defenses
Rachel Greenstadt
April 18, 2012
Reminders

• Project 1 due today (11:59 pm)
• Please send me your project 2 / research groups by the end of this week
• Project 2 due in two weeks
• Next week - research proposal due
Risk Analysis for Software

Common Defense Methods

- Security by Correctness (no more bugs)
  - “Safe” programming languages
  - Code verification
  - Fuzz testing
- Security by Architecture (hard to exploit bugs)
  - Stack protection mechanisms
  - Obfuscation
- Security by Isolation (Sandboxing)
- Security by Response (Patching)
“Safe” Programming Languages

- Just write it in (Java, C#, ML, Haskell, etc)?
- Memory safety - Can’t arbitrarily copy bits from one location to another
- Type Safety “A language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data.”
- Bugs still exist
  - Still a good idea
  - Not all code can be written (or rewritten) in these languages practically
  - Efficiency, low-level access needed, legacy code
What exactly is type safe anyway?

- **Absolute type safety**--type errors are not possible in a well-formed program; the implementation guarantees that every operation on every object will succeed; else the program is not well-formed. Type erasure is possible and is often performed as a runtime optimization. The statically-typed FPLs are in this category.

- **Nominal type safety**--type errors are not possible in a well-formed program, unless explicit "dynamic" constructs (instance_of, dynamic_cast, class introspection) are used. When type errors do occur; the behavior of the system is deterministic and it’s integrity is not compromised. Java 1.4 comes to mind (though not Java 5, due to issues with the generics implementation).

- **Nominal type safety with exceptions**. Type errors are not possible in a well-formed program; unless explicit "type unsafe" operations are performed. Such explicit operations may, if done incorrectly, cause undefined behavior and/or violate the integrity of the programming model. Examples include C#, Eiffel, and Modula 3.

- **Dynamic type safety**. Type errors possible in well-formed programs (and do not require use of explicit constructs which "waive" typechecking). The behavior of the system in the presence of type errors is well-defined; programs can often productively "trap" type errors as a metaprogramming construct. Java 5 is one example; as are most dynamically-typed languages (Smalltalk, Python, Ruby, Lisp).
Manual Code Verification

- A lot of evidence that peer reviews help find bugs, improve software quality
- People write better code if their colleagues are going to look at it
- Teaches people about secure software
- Good code review catches ~50% of bugs [McGraw]
Michael Howard’s 19 Deadly Sins of Software

- Buffer Overflows
- Format String problems
- SQL injection
- Command injection
- Failure to handle errors
- Cross-site scripting
- Failing to protect network traffic
- Use of "magic" URLs and hidden forms
- Improper use of SSL
- Use of weak password-based systems
- Failing to store and protect data
- Information leakage
- Improper file access
- Integer range errors
- Trusting network address information
- Signal race conditions
- Unauthenticated key exchange
- Failing to use cryptographically strong random numbers
- Poor usability
Static Analysis

• Computers better than humans at boring, repetitive tasks?

• Hand - auditing the software you use is too big a task ... sometimes you don’t have code ... can vendors be trusted?

• A lot of this from Tal Garfinkel
Two Types of Static Analysis

• The type you write in 100 lines of python.
  – Look for known unsafe string functions `strncpy()`, `sprintf()`, `gets()`
  – Look for unsafe functions in your source base
  – Look for recurring problem code (problematic interfaces, copy/paste of bad code, etc.)

• The type you get a PhD for
  – Buy this from coverity, fortify, etc.
  – Built into visual studio
  – Roll your own on top of LLVM or Phoenix if hardcore
Static Analysis Basics

• Model program properties abstractly, look for problems
• Tools come from program analysis
  – Type inference, data flow analysis, theorem proving
• Usually on source code, can be on byte code or disassembly
• Strengths
  – Complete code coverage (in theory)
  – Potentially verify absence/report all instances of whole class of bugs
  – Catches different bugs than dynamic analysis
• Weaknesses
  – High false positive rates
  – Many properties cannot be easily modeled
  – Difficult to build
  – Almost never have all source code in real systems (operating system, shared libraries, dynamic loading, etc.)
Example: Where is the bug?

```c
int read_packet(int fd)
{
    char header[50];
    char body[100];
    size_t bound_a = 50;
    size_t bound_b = 100;

    read(fd, header, bound_b);
    read(fd, body, bound_b);

    return 0;
}
```
Example: Where is the bug?

```c
int read_packet(int fd)
{
    char header[50]; //model (header, 50)
    char body[100];  //model (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

    read(fd, header, 100);
    read(fd, body, 100);

    return 0;
}
```
Example: Where is the bug?

```c
int read_packet(int fd)
{
    char header[50]; //model (header, 50)
    char body[100];  //model (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

    read(fd, header, 100); //constant propagation
    read(fd, body, 100);   //constant propagation

    return 0;
}
```
Example: Where is the bug?

```c
int read_packet(int fd)
{
    char header[50]; //model (header, 50)
    char body[100];  //model (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

    //check read(fd, dest.size >= len)
    read(fd, header, 100); //constant propagation
    read(fd, body, 100);    //constant propagation

    return 0;
}
```
Example: Where is the bug?

```c
int read_packet(int fd)
{
    char header[50]; //model (header, 50)
    char body[100];  //model (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

    //check read(fd, 50 >= 100) => SIZE MISMATCH!!
    read(fd, header, 100); //constant propagation
    read(fd, body, 100);   //constant propagation

    return 0;
}
```
Rarely are Things This Clean

- Need information across functions
- Ambiguity due to pointers
- Lack of association between size and data type…
- Lack of information about program inputs/runtime state…
Rarely are Things This Clean

• Need information across functions
• Ambiguity due to pointers
• Lack of association between size and data type…
• Lack of information about program inputs/ runtime state…

Static Analysis is not a panacea, still its very helpful especially when used properly.
Care and Feeding of Static Analysis Tools

• Run and Fix Errors Early and Often
  – otherwise false positives can be overwhelming.

• Use Annotations
  – Will catch more bugs with few false positives

• Write custom rules!
  – Static analysis tools provide institutional memory

• Take advantage of what your compiler provides
  – gcc -Wall, /analyze in visual studio

• Bake it into your build or source control
Normal Dynamic Analysis

• Run program in instrumented execution environment
  – Binary translator, Static instrumentation, emulator
• Look for bad stuff
  – Use of invalid memory, race conditions, null pointer deref, etc.
• Examples: Purify, Valgrind, Normal OS exception handlers (crashes)
Regression vs. Fuzzing

• Regression: Run program on many normal inputs, look for badness.
  – Goal: Prevent normal users from encountering errors (e.g. assertions bad).

• Fuzzing: Run program on many abnormal inputs, look for badness.
  – Goal: Prevent attackers from encountering exploitable errors (e.g. assertions often ok)
Fuzz Testing

- Generate “random” inputs to program
  - Sometimes conforming to input structures (file formats, etc)
- See if program crashes
  - If crashes, found a bug
- Bug may be exploitable
- Surprisingly effective
- Now standard part of development lifecycle, e.g., for IE
Weaknesses of Fuzz Testing

• Probably better for attack than defense
  • Easy to find one or two bugs
  • Hard to find all bugs
  • Still probably a good idea...
Fuzzing Basics

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target interface
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, .mpg)
- Or network based…
  - http, SNMP, SOAP
- Or other…
  - e.g. crashme()
Trivial Example

- Standard HTTP GET request
  - GET /index.html HTTP/1.1

- Anomalous requests
  - AAAAAA...AAAAA /index.html HTTP/1.1
  - GET ///////////////////////////////////////////////////////////////////index.html HTTP/1.1
  - GET %n%n%n%n%n%n.html HTTP/1.1
  - GET /AAAAAAAAAAAAAAA.html HTTP/1.1
  - GET /index.html HTTTTTTTTTTTTTTTTTTP/1.1
  - GET /index.html HTTP/1.1.1.1.1.1.1.1
Different Ways To Generate Inputs

- Mutation Based - “Dumb Fuzzing”
- Generation Based - “Smart Fuzzing”
Mutation Based Fuzzing

- Little or no knowledge of the structure of the inputs is assumed

- Anomalies are added to existing valid inputs

- Anomalies may be completely random or follow some heuristics (e.g. remove NUL, shift character forward)

- Examples:
  - Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.
Example: fuzzing a pdf viewer

- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script to)
  1. Grab a file
  2. Mutate that file
  3. Feed it to the program
  4. Record if it crashed (and input that crashed it)
Dumb Fuzzing In Short

• Strengths
  – Super easy to setup and automate
  – Little to no protocol knowledge required

• Weaknesses
  – Limited by initial corpus
  – May fail for protocols with checksums, those which depend on challenge response, etc.
Generation Based Fuzzing

- Test cases are generated from some description of the format: RFC, documentation, etc.
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing
Example: Protocol Description

//png.spk
//@author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
    s_string("IHDR"); // type
    s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYTE
    s_push_int(0x1a, 1); // Width
    s_push_int(0x14, 1); // Height
    s_push_int(0x8, 3); // Bit Depth - should be 1,2,4,8,16, based on colortype
    s_push_int(0x3, 3); // ColorType - should be 0,2,3,4,6
    s_binary("00 00"); // Compression || Filter - shall be 00 00
    s_push_int(0x0, 3); // Interlace - should be 0,1

    s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
    s_block_end("IHDRcrc");
...
Generation Based Fuzzing In Short

- **Strengths**
  - completeness
  - Can deal with complex dependencies e.g. checksums

- **Weaknesses**
  - Have to have spec of protocol
    - Often can find good tools for existing protocols e.g. http, SNMP
  - Writing generator can be labor intensive for complex protocols
  - The spec is not the code
Input Generation

• Existing generational fuzzers for common protocols (ftp, http, SNMP, etc.)
  – Mu-4000, Codenomicon, PROTOS, FTPFuzz
• Fuzzing Frameworks: You provide a spec, they provide a fuzz set
  – SPIKE, Peach, Sulley
• Dumb Fuzzing automated: you provide the files or packet traces, they provide the fuzz sets
  – Filep, Taof, GPF, ProxyFuzz, PeachShark
• Many special purpose fuzzers already exist as well
  – ActiveX (AxMan), regular expressions, etc.
How Much Fuzz Is Enough?

• Mutation based fuzzers can generate an infinite number of test cases... When has the fuzzer run long enough?

• Generation based fuzzers generate a finite number of test cases. What happens when they’re all run and no bugs are found?
Example: PDF

- I have a PDF file with 248,000 bytes
- There is one byte that, if changed to particular values, causes a crash
  - This byte is 94% of the way through the file
- Any single random mutation to the file has a probability of 0.00000392 of finding the crash
- On average, need 127,512 test cases to find it
- At 2 seconds a test case, that's just under 3 days...
- It could take a week or more...
Code Coverage

• Some of the answers to these questions lie in code coverage
• Code coverage is a metric which can be used to determine how much code has been executed.
• Data can be obtained using a variety of profiling tools. e.g. gcov
Types of Code Coverage

• Line coverage
  – Measures how many lines of source code have been executed.

• Branch coverage
  – Measures how many branches in code have been taken (conditional jmps)

• Path coverage
  – Measures how many paths have been taken
Example

if( a > 2 )
a = 2;
if( b > 2 )
b = 2;

• Requires
  – 1 test case for line coverage
  – 2 test cases for branch coverage
  – 4 test cases for path coverage
    • i.e. \((a, b) = \{(0, 0), (3, 0), (0, 3), (3, 3)\}\)
Problems with Code Coverage

- Code can be covered without revealing bugs
  ```c
  mySafeCpy(char *dst, char* src){
      if(dst && src)
          strcpy(dst, src);
  }
  ```

- Error checking code mostly missed (and we don’t particularly care about it)
  ```c
  ptr = malloc(sizeof(blah));
  if(!ptr)
      ran_out_of_memory();
  ```

- Only “attack surface” reachable
  - i.e. the code processing user controlled data
  - No easy way to measure the attack surface
    - Interesting use of static analysis?
Code Coverage Good For Lots of Things

• How good is this initial file?
• Am I getting stuck somewhere?

```java
if(packet[0x10] < 7) { //hot path
} else { //cold path
}
```

• How good is fuzzer X vs. fuzzer Y
• Am I getting benefits from running a different fuzzer?

See Charlie Miller’s work for more!
Also FX
Fuzzing Rules of Thumb

- Protocol specific knowledge very helpful
  - Generational tends to beat random, better spec’s make better fuzzers
- More fuzzers is better
  - Each implementation will vary, different fuzzers find different bugs
- The longer you run, the more bugs you find
- Best results come from guiding the process
  - Notice where you're getting stuck, use profiling!
- Code coverage can be very useful for guiding the process
The Future of Fuzz
Outstanding Problems

• What if we don’t have a spec for our protocol/How can we avoid writing a spec.

• How do we select which possible test cases to generate
Whitebox Fuzzing

- Infer protocol spec from observing program execution, then do generational fuzzing
- Potentially best of both worlds
- Bleeding edge
How do we generate constraints?

• Observe running program
  – Instrument source code (EXE)
  – Binary Translation (SAGE, Catchconv)
• Treat inputs as symbolic
• Infer constraints
Example:

```c
int test(x)
{
    if (x < 10) {
        // X < 10 and X <= 0 gets us this path
        if (x > 0) {
            // 0 < X < 10 gets us this path
            return 1;
        }
    }
    // X >= 10 gets us this path
    return 0;
}
```

Constraints:

- $X \geq 10$
- $0 < X < 10$
- $X \leq 0$

Solve Constraints -- we get test cases: {12,0,4}

- Provides maximal code coverage
Greybox Techniques

- Evolutionary Fuzzing
- Guided mutations based on fitness metrics
- Prefer mutations that give
  - Better code coverage
  - Modify inputs to potentially dangerous functions (e.g. memcpy)
- EFS, autodafe
Summary

• To find bugs, use the tools and tactics of an attacker
• Fuzzing and static analysis belong in every developers toolbox
• Field is rapidly evolving
• If you don’t apply these tools to your code, someone else will
Limitations of Correctness?
Security Through Architecture

- Stackguard, LibSafe, Pointguard
- Stack randomization
- Non-executable Stack
- Obfuscation

- Why do some hackers call this security through obscurity?
Run time checking:

- Many many run-time checking techniques …
- We only discuss methods relevant to overflow protection

**Solution 1: StackGuard**

- Run time tests for stack integrity.
- Embed “canaries” in stack frames and verify their integrity prior to function return.
Canary Types

• **Random canary:**
  - Choose random string at program startup.
  - Insert canary string into every stack frame.
  - Verify canary before returning from function.
  - To corrupt random canary, attacker must learn current random string.

• **Terminator canary:**
  - Canary = 0, newline, linefeed, EOF
  - String functions will not copy beyond terminator.
  - Attacker cannot use string functions to corrupt stack.
StackGuard (Cont.)

• StackGuard implemented as a GCC patch.
  • Program must be recompiled.

• Minimal performance effects: 8% for Apache.

• Note: Canaries don’t offer fullproof protection.
  • Some stack smashing attacks leave canaries unchanged

• Heap protection: PointGuard.
  • Protects function pointers and setjmp buffers by encrypting them: XOR with random cookie

• More noticeable performance effects
Run time checking: Libsafe

- **Solution 2**: Libsafe (Avaya Labs)
  - Dynamically loaded library.
  - Intercepts calls to `strcpy(dest, src)`
  - Validates sufficient space in current stack frame:
    \[ |\text{frame-pointer} – \text{dest}| > \text{strlen(src)} \]
  - If so, does `strcpy`, otherwise, terminates application
Non-Executable Stack

- NX bit on every Page Table Entry
- AMD Athlon 64, Intel P4 “Prescott”
- Code patches marking stack segment as non-executable exist for Linux, Solaris, OpenBSD
- Some applications need executable stack
  - For example, LISP interpreters
- Does not defend against return-to-libc exploits
  - Overwrite return address with the address of an existing library function (can still be harmful)
ASLR

- Address Space Layout Randomization
- Randomize place where code is put into memory
- Makes most buffer overflow attacks probabilistic
- Vista does this (256 random layouts)
  - Similar thing is done in Mac OS X
Security by Response

• Group exercise:
  • What issues must be considered in releasing a patch for a vulnerability?
  • What about applying a patch?
Vulnerability Analysis and Disclosure

• What do you do if you’ve found a security problem in a real system?
  • IM client?
  • Electronic voting machine?
  • ATM machine?
  • Hospital drug (morphine) pump?
Monoculture

• How does software monoculture affect security?