Tales of Woe:

Seven Deadly Vulnerabilities

GHOST ● Heartbleed ● Conficker ● Stagefright ● Shellshock ● Java Deserialization ● VENOM
Tale #1: GHOST (Gnu HOST bug)

- Bug in the Linux glibc library
- Discovered by Qualys researchers during a routine code audit in 2015
- Affects all code that uses glibc for host-lookups (i.e., nearly all Linux networking software) between 2000-2013
- Can you spot the bug?

```c
int __nss_hostname_digits_dots( ... ) {
    ...
    size_needed = sizeof(*host_addr) + sizeof(*h_addr_ptrs) + strlen(name) + 1;
    *buffer = (char*) malloc(size_needed);

    ... 35 lines of code ...

    host_addr = (host_addr_t*) *buffer;
    h_addr_ptrs = (host_addr_list_t*) ((char*) host_addr + sizeof(*host_addr));
    h_alias_ptr = (char**) ((char*) h_addr_ptrs + sizeof(*h_addr_ptrs));
    hostname = (char*) h_alias_ptr + sizeof(*h_alias_ptr);
    ...
```
Tale #1: GHOST (Gnu HOST bug)

- Bug in the Linux glibc library
- Discovered by Qualys researchers during a routine code audit in 2015
- Affects all code that uses glibc for host-lookups (i.e., nearly all Linux networking software) between 2000-2013
- Can you spot the bug?

```c
1 int __nss_hostname_digits_dots( ... ) {
  ...

3  size_needed = sizeof(*host_addr) + sizeof(*h_addr_ptrs) + strlen(name) + 1;
4  *buffer = (char*) malloc(size_needed);

... 35 lines of code ...

5  host_addr = (host_addr_t*) *buffer;
6  h_addr_ptrs = (host_addr_list_t*) ((char*) host_addr + sizeof(*host_addr));
7  h_alias_ptr = (char**) ((char*) h_addr_ptrs + sizeof(*h_addr_ptrs));
8  hostname = (char*) h_alias_ptr + sizeof(*h_alias_ptr);

...
Is it really that big a deal?

Qualys was able to take complete remote control of affected Linux machines merely by sending them a maliciously crafted email (unread!).

Can you figure out how they did it?
Is it really that big a deal?

Qualys was able to take complete remote control of affected Linux machines merely by sending them a maliciously crafted email (unread!).

Can you figure out how they did it?
Tale #2: Heartbleed

- Bug in the OpenSSL (secure web communications!) library discovered by Codenomicon in 2014
- Buffer over-read error in implementation of Heartbeat TLS protocol:
  - read-loop trusts length bound provided by user
  - over-read data sent directly back to attacker
- Vulnerability exposed ~66% of the internet to theft of encryption keys between 2011-2014.
- Still highly exploitable because OpenSSL is so pervasive, cannot always be patched in the wild.
- Heartbeat packets deemed so innocuous, they were not even logged during the zero-day window.

```c
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0];
    unsigned int payload;
    n2s(p, payload);
    ...
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, p, payload);
    bp += payload;
    ...
}
Tale #2: Heartbleed

- Bug in the OpenSSL (secure web communications!) library discovered by Codenomicon in 2014
- Buffer over-read error in implementation of Heartbeat TLS protocol:
  - read-loop trusts length bound provided by user
  - over-read data sent directly back to attacker
- Vulnerability exposed ~66% of the internet to theft of encryption keys between 2011-2014.
- Still highly exploitable because OpenSSL is so pervasive, cannot always be patched in the wild.
- Heartbeat packets deemed so innocuous, they were not even logged during the zero-day window.

```c
int dtls1_process_heartbeat(SSL *s) {
  unsigned char *p = &s->s3->rrec.data[0];
  unsigned int payload;
  n2s(p, payload);
  ...
  buffer = OPENSSL_malloc(1 + 2 + payload + padding);
  bp = buffer;
  *bp++ = TLS1_HB_RESPONSE;
  s2n(payload, bp);
  memcpy(bp, p, payload);
  bp += payload;
  ...
```
Tale #3: MS08-067 (Conficker Exploit)

- Bug in Windows netapi32.dll lib first discovered in 2008
- Allows complete remote compromise of all (then) Windows Servers
- Exploited by Confiker worm to infect ~1.7 million machines in ~190 countries, including national defense networks across Europe

```c
void _NetpwPathCanonicalize(wchar_t* Path) {
    if (!_function_check_length(Path)) return;
    ...
    _CanonicalizePathName(Path);
    ...
}

void _CanonicalizePathName(wchar_t* Path) {
    wchar_t wcsBuffer[0x420];
    ...
    wcscat(wcsBuffer, Path);
    ...
    _ConvertPathMacros(wcsBuffer);
    ...
```
Tale #3: MS08-067 (Conficker Exploit)

- Bug in Windows netapi32.dll lib first discovered in 2008
- Allows complete remote compromise of all (then) Windows Servers
- Exploited by Confiker worm to infect ~1.7 million machines in ~190 countries, including national defense networks across Europe

```c
void _NetpwPathCanonicalize(wchar_t* Path) {
    if (!_function_check_length(Path)) return;
    
    _CanonicalizePathName(Path);
    
}

void _CanonicalizePathName(wchar_t* Path) {
    wchar_t wcsBuffer[0x420];
    
    wcscat(wcsBuffer, Path);
    
    _ConvertPathMacros(wcsBuffer);
    
```
Tale #4: Stagefright

- Series of 8 critical vulnerabilities discovered in Android OS 2014-2015
- Allows complete remote hijacking of 95% of Android devices
- No user interaction required! (merely receiving a malformed MMS message triggers bug)

```c
status_t SampleTable::setTimeToSampleParams(...) {
    uint32_t mTimeToSampleCount = U32_AT(&header[4]);
    uint64_t allocSize = mTimeToSampleCount * 2 * sizeof(uint32_t);
    if (allocSize > SIZE_MAX) return ERROR_OUT_OF_RANGE;
    mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
    ...
```
Tale #4: Stagefright

- Series of 8 critical vulnerabilities discovered in Android OS 2014-2015
- Allows complete remote hijacking of 95% of Android devices
- No user interaction required! (merely receiving a malformed MMS message triggers bug)

```c
status_t SampleTable::setTimeToSampleParams(...) {
    uint32_t mTimeToSampleCount = U32_AT(&header[4]);
    uint64_t allocSize = mTimeToSampleCount * 2 * sizeof(uint32_t);
    if (allocSize > SIZE_MAX) return ERROR_OUT_OF_RANGE;
    mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
    ...
```
Tale #5: Shellshock (Linux Bash Bug)

- Bug (undocumented feature?) discovered in Linux bash shell (by IT manager Stephane Chazelas in his spare time!) in 2014
- Bash command-parser interprets certain text in environment variables as code and executes it during parsing(?!)
- Impact: All Linux software storing user-provided data in environment variables susceptible to complete remote compromise.
- Zero-day window: 25 years(!!) (198?-2014)

```c
void initialize_shell_variables(char **env, int privmode) {
    ...
    for (string_index = 0; string = env[string_index++]; ) {
        ...
        if (privmode==0 && read_but_dont_execute == 0 && STREQN("() {", string, 4)) {
            ...
            parse_and_execute(temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);
        ...
    }
```
Tale #6: Java Deserialization

- Logical flaw in how many Java applications receive objects as input
- Examples dating back to 2010 and before, but popularized in 2015-2018 by successful attacks against WebSphere, WebLogic, JBoss, etc. [FoxGlove’15]
- Millions of Java apps estimated to be currently vulnerable to complete remote compromise

The Problem:
- Java apps must deserialize input stream to object before they know what kind of object they received.
- JVM deserializes stream to whatever object it says it is.
- Some built-in JVM objects execute code at object initialization.
- Executed code is supplied by attacker!
Tale #7: VENOM (Virtualized Environment Neglected Operations Manipulation)

- floppy disk controller bug discovered in 2015
- affects many VMs and hypervisors: QEMU, Xen, KVM, VirtualBox, ...
- allows guest OS to escape the VM sandbox and run code on the host
- millions of data centers at risk
- existed for 10 years(!) before patched
- buffer overwrite error

```c
void fdctrl_write_data(FDCtrl *fdctrl, uint32_t value) {
    ...
    fdctrl->fifo[fdctrl->data_pos++] = value;
    ...
}```
The Software Security Crisis

MITRE CVE Top “Unforgivable Vulnerabilities”
- buffer overflow
- XSS
- SQL injection
- directory traversal
- world-writable files
- direct admin script requests
- homegrown crypto
- authentication bypass
- large check-use windows (TOCTOU)
- privilege escalation
- undocumented account
- integer overflow

Why do these still occur? Why do standard approaches fail?
Misguided Solutions

- People who haven’t studied the field think the solution is “obvious”:
  - Naïve idea #1: “If everyone just used [ Linux | Java | Mac | ... ]”
  - Naïve idea #2: “Stop hiring stupid programmers.”
  - Naïve idea #3: “Prioritize security testing more. Don’t release too soon.”
  - Naïve idea #4: “Just configure your permissions properly.”

- IT approaches today:
  - Patch early, patch often...
  - Monitor network packets, monitor syscalls, monitor phone calls (NSA)...
  - Penetration testing (red-teaming)
  - Source code review
Science of Software Security

- **Goals**
  - Find long-term, universal solutions to software security crisis
  - Obtain mathematical, quantifiable guarantees for security of software products
    - machine-checked proofs, reliable metrics
  - Automate rigorous checking processes
    - no human in the loop!
- **Two main domains of research**
  - new languages/tools for creating secure software from scratch
  - securing legacy code
- **Three stages of enforcement**
  - static (find & fix vulnerabilities before runtime)
  - dynamic (detect and block attacks at runtime)
  - audit (recover and assign blame after an attack)
Important LBS Technologies

- Automated theorem-provers
  - machine-assisted, machine-checked proofs of security
- In-lined Reference Monitors
  - insert dynamic security checks into untrusted code
- Type-checkers
  - advanced type systems can encode security properties
- Model-checkers
  - statically verify that code model obeys a security property
- Certifying Compilers
  - transform source code into object code and an independently verifiable proof that the object code is safe to execute
At Least Three Hard Issues Involved

- Minimal Trusted Computing Base (TCB)
- Principle of Least Privilege
- The Model Problem:
  - Trust Model
  - Attacker Model
  - System Model
Let’s play a game: I’m thinking of a piece of software.

- Most of you have it and have used it.
- If it fails, it could delete or divulge all your personal files.
- Microsoft makes it.
- Can you guess which software I’m thinking of?
Let’s play a game: I’m thinking of a piece of software.

- Most of you have it and have used it.
- If it fails, it could delete or divulge all your personal files.
- Microsoft makes it.
- Can you guess which software I’m thinking of?
Least Privilege

- **Principle of Least Privilege:** “Every program and every user of the system should operate using the least set of privileges necessary to complete the job.” [Saltzer & Schroeder, 1975]

- Hard problem: What is the least set of privileges necessary to complete the job? How do we compute it?

- No finite set of roles or permission options suffices to meet PoLP in all cases!

- Richer classes of enforceable policies get us closer, though.
Trust Modeling

Client (web browser) ➔ Web Server

URL Request
Trust Modeling

Client (web browser) 

web page 

Web Server
Trust Modeling

Client (web browser) → Ad request → Ad Network → Web Server
Trust Modeling

Client (web browser) → ad URL → Web Server → Ad Network
Trust Modeling

Client (web browser) → URL request → Ad Network → Ad Server → Web Server
Trust Modeling

Client (web browser) → Ad Network → Ad Server → Web Server
Trust Modeling

- Four principals: client, page publisher, ad network, ad publisher
- What are some security requirements each principal is likely to have?
- Which existing technologies can be used to meet those requirements?
- How can we assess/measure the “security” of the resulting system?
Trust Modeling

- Trust model: Who trusts whom to do what?
- Trusted Computing Base (TCB): The set of all system components that must be trusted in order to maintain system security
  - Security meta-goal: minimize the TCB
- What is the trust model in our web scenario?
- What is the TCB? How can we make it smaller?
Attack Modeling

- Threat model: set of assumed attacker capabilities
  - attacks outside the model may succeed!
  - threat model assumptions = security system limitations
- What is a reasonable threat model for our web scenario?
Major Classes of Security Policies

- **Integrity** - preventing improper or unauthorized change to data or resources
  - Example: ad may not delete your files
- **Availability** - continued access to data or resources
  - Example: ad may not expand to occlude the rest of the page
- **Confidentiality** - concealment of data or resources
  - Example: ad may not send your browsing history to your employer
Defining Security Policies Formally

- **Security Policy** - specification of allowed (or, equivalently, disallowed) behaviors
  - **Safety Policies** - some “bad” thing shouldn’t happen (integrity)
  - **Liveness Policies** - some “good” thing should eventually happen (availability)
- **Safety + Liveness = all policies** [Alpern & Schneider, 1985]
Software Lifecycle

- Security vulnerabilities in non-malicious code
  - type-safe programming languages
  - formal verification
  - code synthesis
- Malicious code (viruses, worms, etc.)
- Antivirus scanning
- Code-signing
- Type-safe target codes (e.g., Java bytecode)
- Independently verifiable certificates
- Runtime monitoring
- Automatically generated self-monitoring code
- Auditing (logging)
- Rollback (reversible computation, restore points)
- Legal action
Example: Memory Safety

- Memory Safety = ?
- Traditional security model:
  - program is a black box
  - OS/hardware intercepts every memory access
- Language-based security model:
  - program is a sequence of instructions in an architecture with known semantics
  - analyze the sequence to identify all potential violations
  - insert dynamic memory checks into the program
Example: Memory Safety

- **Memory Safety** = Programs may not access unallocated memory addresses
- **Traditional security model:**
  - program is a black box
  - OS/hardware intercepts every memory access
- **Language-based security model:**
  - program is a sequence of instructions in an architecture with known semantics
  - analyze the sequence to identify all potential violations
  - insert dynamic memory checks into the program
Example: Data Confidentiality

- Policy: Don’t divulge my credit card number
- Traditional approach:
  - monitor all outgoing network traffic
  - block any transmission containing the relevant bit sequence
- Language-based approach:
  - analyze the dataflow graph of the software
  - identify potential flows from high-security sources to low-security sinks
  - interpose robust declassification guards along identified flows
  - quantify the potential information disclosure as Shannon entropy
Reasons for a Language-based Approach

- **Rigor**
  - We have a science of programming languages!
  - Lets us prove things about how software behaves and what it can do

- **Efficiency**
  - enforce security “from inside” the software
  - richer context, smarter security checks, fewer context switches

- **Flexibility**
  - no need for custom OS/hardware
  - ship the enforcement mechanism with the product, or add it client-side

- **Power/expressiveness**
  - can enforce exceptionally powerful policies (e.g., history-based)
  - enforce notoriously hard policies like confidentiality and availability
Decidability

- Is this really possible with arbitrary software? What about these guys?
- The Halting Problem
  - Exercise: Reduce memory safety to the halting problem
- Escape Hatches
  - conservative rejection
  - limit the domain (e.g., constrained input language)
  - require dynamic checks on uncertainty
  - push the proof burden to the code-provider
Next Time: Software Model Checking

- Software Model Checking vs. Automated Theorem Proving

- Lists assignment also due Monday
  - Be sure you have at least a tentative solution to \texttt{matches\_nil} and \texttt{rem} from Assignment 1 (even if they might have bugs).
  - probably easier than last two assignments if you have a mostly-correct Assignment 1 (but don’t wait until the last second!)