E-VOTING ANALYSIS

Kohno et al., IEEE S&P 2004
Halderman, 2016
“Security mindset”

• Consider a complex system:
  • Potential security threats?
  • Hidden and explicit assumptions
  • How to mitigate the risks?
  • What are different players’ incentives?
1. Summarize the system

1. Pre-election: Poll worker loads “ballot definition” via e.g. USB

2. Voting: Voter obtains single-use smartcard, votes, vote stored encrypted, card canceled

3. Post-election: Votes decrypted and sent to tabulator, who counts
2. Identify goals/requirements

• **Confidentiality**: Can’t find out who I voted for

• **Integrity**: Can’t alter votes

• **Availability**: Can’t deny opportunity to vote

• **Usability**: General public can vote correctly without undue burden

What if the attacker can violate these, but you catch him/her?
3. Identify adversaries/threats

- Poll worker, voter, outsider
- Display one vote / count a different vote
- Vote multiple times
- End election early (DOS)
- Tamper with stored data
- Reveal who voted for whom
Diebold Accuvote TS

- Used in 37 states! (in 2004)
- No cryptography protects smartcards, ballot definition file
- “Protected counter” in single, mutable file
- Pose as voting machine, send to tabulator
- Homebrew crypto protects vote logs  
  - Hardcoded key since at least 1998
- Read the paper for more
Follow-up

• More researchers confirmed these bugs and found others (got real hardware)
• State investigations: MD, CA, OH
  • Similar problems from other manufacturers
  • Sequoia AVC: designed 1980, used in NJ 2009
• “By the 2014 general election, 70% of American voters were casting ballots on paper”
Takeaways

• Adversarial thinking
• Whole-systems view
  • Hardware, software, network, users, economics
• Only as strong as weakest link
  • Break into building vs. sniff unencrypted traffic
  • You have to be right always, adversary once
• Never homebrew crypto!
• Security through obscurity DOESN’T WORK!
This time

We will begin our 1st section: **Software Security**

By investigating **Buffer overflows** and other memory safety vulnerabilities

- History
- Memory layouts
- Buffer overflow fundamentals
screensaver --prompt="Don't unlock plz"

Don't unlock plz

Locked by dml
press ctrl-c to logout
screensaver --prompt="Don’t unlock pretty plz"

Don't unlock pretty plz

Locked by dml
press ctrl-c to logout
screensaver --prompt="Don’t unlock plz"

Don't unlock plz

Locked by dml

press ctrl-c to logout
screensaver --prompt="Under maintenance; 
Do not interrupt

Locked by dml
press ctrl-c to logout
Most (interesting) software takes input

Target host (victim)

Direct user interaction
- command line interface (stdin)
- user opens a document
Most (interesting) software takes input

Target host (victim)

Direct user interaction
- command line interface (stdin)
- user opens a document

Network communication
- emails
- various protocols
Most (interesting) software takes input

Target host (victim)

- Direct user interaction
  - command line interface (stdin)
  - user opens a document

- Network communication
  - emails
  - various protocols

- Sensing the outside world
  - QR codes (to link w/ malware)
  - sound recordings

- Third-party libraries
- Future code updates
- Others…

Goal: Correct operation despite malicious inputs
What is a buffer overflow?

• A **low-level** bug, typically in **C/C++**
  • Significant security implications!

• If accidentally triggered, causes a crash

• If maliciously triggered, can be **much worse**
  • **Steal** private info
  • **Corrupt** important info
  • **Run** arbitrary code
Why study them?

• Buffer overflows are still relevant today
  • C and C++ are still popular
  • Buffer overflows still occur with regularity

• They have a long history
  • Many different approaches developed to defend against them, and bugs like them

• They share common features with other bugs we will study
  • In how the attack works
  • In how to defend against it
C and C++ still very popular

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. C</td>
<td>📡📱🔋</td>
<td>100.0</td>
</tr>
<tr>
<td>2. Java</td>
<td>🌐📱📱</td>
<td>98.1</td>
</tr>
<tr>
<td>3. Python</td>
<td>🌐💻📱</td>
<td>98.0</td>
</tr>
<tr>
<td>4. C++</td>
<td>📡📱🔋</td>
<td>95.9</td>
</tr>
<tr>
<td>5. R</td>
<td>📡💻</td>
<td>87.9</td>
</tr>
<tr>
<td>6. C#</td>
<td>🌐📱📱</td>
<td>86.7</td>
</tr>
<tr>
<td>7. PHP</td>
<td>🌐💻</td>
<td>82.8</td>
</tr>
<tr>
<td>8. JavaScript</td>
<td>🌐📱</td>
<td>82.2</td>
</tr>
<tr>
<td>9. Ruby</td>
<td>🌐💻📱</td>
<td>74.5</td>
</tr>
<tr>
<td>10. Go</td>
<td>🌐💻📱</td>
<td>71.9</td>
</tr>
</tbody>
</table>

http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages
Critical systems in C/C++

- Most **OS kernels** and utilities
  - fingerd, X windows server, shell

- Many **high-performance servers**
  - Microsoft IIS, Apache httpd, nginx
  - Microsoft SQL server, MySQL, redis, memcached

- Many **embedded systems**
  - Mars rover, industrial control systems, automobiles, healthcare devices

A successful attack on these systems is particularly dangerous!
We’re going to focus on C

A breeding ground for buffer overflow attacks

• Morris worm
  • Propagated across machines (too aggressively, thanks to a bug)
  • One way it propagated was a buffer overflow attack against a vulnerable version of fingerd on VAXes
    • Sent a special string to the finger daemon, which caused it to execute code that created a new worm copy
    • Didn’t check OS: caused Suns running BSD to crash
  • End result: $10-100M in damages, probation, community service

(Robert Morris is now a professor at MIT)
We’re going to focus on C

A breeding ground for buffer overflow attacks

- CodeRed
  - Exploited an overflow in the MS-IIS server
  - 300,000 machines infected in 14 hours
We’re going to focus on C

A breeding ground for buffer overflow attacks

- SQL Slammer
  - Exploited an overflow in the MS-SQL server
  - 75,000 machines infected in 10 minutes
We’re going to focus on C

A breeding ground for buffer overflow attacks


• Conficker worm
  • Exploited an overflow in Windows RPC
  • ~10 million machines infected
We’re going to focus on C

A breeding ground for buffer overflow attacks

- Stuxnet
  - Exploited several *overflows* nobody had at the time known about ("zero-day")
    - Windows print spooler service
    - Windows LNK shortcut display
    - Windows task scheduler
  - Also exploited the same Windows RPC overflow as Conficker
  - Impact: legitimized cyber warfare (*more on this later*)
We’re going to focus on C

A breeding ground for buffer overflow attacks

- Flame
  - Same print spooler and LNK overflows as Stuxnet
  - Cyber-espionage virus
23-Year-Old X11 Server Security Vulnerability Discovered

Posted by Unknown Lamer on Wednesday, January 08, 2014 @10:11a.m.
from the stack-smashing-for-fun-and-profit dept.

An anonymous reader writes

"The recent report of X11/X.Org security in bad shape rings more truth today. The X.Org Foundation announced today that they've found a X11 security issue that dates back to 1991. The issue is a possible stack buffer overflow that could lead to privilege escalation to root and affects all versions of the X Server back to X11R5. After the vulnerability being in the code-base for 23 years, it was finally uncovered via the automated cppcheck static analysis utility."

There's a scanf used when loading BDF fonts that can overflow using a carefully crafted font. Watch out for those obsolete early-90s bitmap fonts."
GHOST: glibc vulnerability introduced in 2000, only just announced last year
syslogd bug in Mac OS X & iOS

- syslog: message logging infrastructure
  - Useful: one process issues the log messages, syslogd handles storing/disseminating them

```c
void add_lockdown_session(int fd)
{
    dispatch_once(&watch_init_once, ^{
        watch_queue = dispatch_queue_create("Direct Watch Queue", NULL);
    });

    dispatch_async(watch_queue, ^{
        if (global.lockdown_session_count == 0) global.lockdown_session_fds = NULL;

        global.lockdown_session_fds = realloc(global.lockdown_session_fds,
                                             global.lockdown_session_count + 1 * sizeof(int));

        if (global.lockdown_session_fds == NULL)
        {
            asldebug("add_lockdown_session: realloc failed\n");
            global.lockdown_session_count = 0;
        }
        else
        {
            global.lockdown_session_fds[global.lockdown_session_count++] = fd;
        }

        global.watchers_active = direct_watch_count + global.lockdown_session_count;
    });
}
syslogd bug in Mac OS X & iOS

- syslog: message logging infrastructure
  - Useful: one process issues the log messages, syslogd handles storing/disseminating them

```c
void add_lockdown_session(int fd)
{
    dispatch_once(&watch_init_once, ^{
        watch_queue = dispatch_queue_create("Direct Watch Queue", NULL);
    });

    dispatch_async(watch_queue, ^{
        if (global.lockdown_session_count == 0) global.lockdown_session_fds = NULL;
        global.lockdown_session_fds = realloc(global.lockdown_session_fds,
                                              global.lockdown_session_count + 1 * sizeof(int));

        if (global.lockdown_session_fds == NULL)
        {
            asldebug("add_lockdown_session: realloc failed\n");
            global.lockdown_session_count = 0;
        } else
        {
            global.lockdown_session_fds[global.lockdown_session_count++] = fd;
        }

        global.watchers_active = direct_watch_count + global.lockdown_session_count;
    });
}
```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds, 
    global.lockdown_session_count + 1 * sizeof(int));
global.lockdown_session_fds = reallocf(global.lockdown_session_fds, 
    global.lockdown_session_count + 1 * sizeof(int));
global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
global.lockdown_session_count + 1 * sizeof(int));

Array of int's

Had this many int's
global.lockdown_session_fds = reallocf(global.lockdown_session_fds, 
               global.lockdown_session_count + 1 * sizeof(int));

Array of **int**'s

Want this many **int**’s
global.lockdown_session_fds = reallocf(global.lockdown_session_fds, global.lockdown_session_count + 1 * sizeof(int));

Takes \textit{bytes} as 2nd arg

Array of \texttt{int}'s

Want this many \texttt{int}'s
Takes bytes as 2nd arg

```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds, 
global.lockdown_session_count + 1 * sizeof(int));
```

Array of \texttt{int}'s

Want this many \texttt{int}'s

How many \texttt{bytes} should \texttt{global.lockdown_session_fds} be?
How many bytes should `global.lockdown_session_fds` be?

```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds, 
    global.lockdown_session_count + 1 * sizeof(int));
```

Array of `int`'s

Want this many `int`'s

Takes `bytes` as 2nd arg

```
global.lockdown_session_count + 1 * sizeof(int)
```

How many `bytes` should `global.lockdown_session_fds` be?
Want this many int's

Array of int's

How many bytes should global.lockdown_session_fds be?

\[
\text{global.lockdown_session_fds} = \text{reallocf}(	ext{global.lockdown_session_fds}, \text{global.lockdown_session_count} + 1 \times \text{sizeof(int)});
\]

\[
\text{global.lockdown_session_count} + 1 \times \text{sizeof(int)}
\]

\[
(\text{global.lockdown_session_count} + 1) \times \text{sizeof(int)}
\]
syslogd bug in Mac OS X & iOS

- syslog: message logging infrastructure
  - Useful: one process issues the log messages, syslogd handles storing/disseminating them

```c
void add_lockdown_session(int fd)
{
    dispatch_once(&watch_init_once, ^{
        watch_queue = dispatch_queue_create("Direct Watch Queue", NULL);
    });

    dispatch_async(watch_queue, ^{
        if (global.lockdown_session_count == 0) global.lockdown_session_fds = NULL;
        global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                                  global.lockdown_session_count + 1 * sizeof(int));

        if (global.lockdown_session_fds == NULL)
        {
            asldebug("add_lockdown_session: realloc failed\n");
            global.lockdown_session_count = 0;
        }
        else
        {
            global.lockdown_session_fds[global.lockdown_session_count++] = fd;
        }
    
    global.watchers_active = direct_watch_count + global.lockdown_session_count;
}
```
syslogd bug in Mac OS X & iOS

- syslog: message logging infrastructure
- Useful: one process issues the log messages, syslogd handles storing/disseminating them

```c
void add_lockdown_session(int fd)
{
    dispatch_once(&watch_init_once, ^{
        watch_queue = dispatch_queue_create("Direct Watch Queue", NULL);
    });

    dispatch_async(watch_queue, ^{
        if (global.lockdown_session_count == 0) global.lockdown_session_fds = NULL;

        global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                                global.lockdown_session_count + 1 * sizeof(int));

        if (global.lockdown_session_fds == NULL)
            asldebug("add_lockdown_session: realloc failed\n");
        global.lockdown_session_count = 0;
    } else
    {
        global.lockdown_session_fds[global.lockdown_session_count++] = fd;
    }

global.watchers_active = direct_watch_count + global.lockdown_session_count;
```
Buffer overflows are prevalent

Significant percent of all vulnerabilities

Data from the National Vulnerability Database
Buffer overflows are prevalent

Total number of buffer overflow vulnerabilities

Data from the National Vulnerability Database
Buffer overflows are impactful

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>[12]</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
</tbody>
</table>

MITRE's top-25 most dangerous software errors (from 2011)
Note about terminology

- We will use **buffer overflow** to mean *any access of a buffer outside of its allotted bounds*
  - An over-read, or an over-write
  - During *iteration* (“running off the end”) or by *direct access*
  - Could be to addresses that *precede* or *follow* the buffer

- Other terms you may hear (more specific)
  - *Underflow*, over-read, out-of-bounds access, etc.
  - Some use *buffer overflow* only for writing off the end
Memory layout
All programs are stored in memory

4G

0

0xffffffff

0x00000000
All programs are stored in memory

The *process’s view* of memory is that it owns all of it

In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses
The instructions themselves are in memory

4G

0x00000000

0xffffffff

0xffffffff

...  
0x4c2 sub $0x224,%esp  
0x4c1 push %ecx  
0x4bf mov %esp,%ebp  
0x4be push %ebp  
...
Data’s location depends on how it’s created

---

4G

0xffffffff

Uninit’d data

Init’d data

Text

0

0x00000000

static int x;

static const int y=10;
Data’s location depends on how it’s created

Known at compile time

Uninit’d data

Init’d data

Text

static int x;
static const int y=10;

0xffffffff

0x00000000

4G
Data’s location depends on how it’s created

Set when process starts

Known at compile time

4G

cmdline & env

Uninit’d data

Init’d data

Text

0xffffffff

0x000000000000

static int x;

static const int y=10;
Data’s location depends on how it’s created

- **Set when process starts**
  - cmdline & env
  - Stack
  - Heap
  - Uninit’d data
  - Init’d data
  - Text

- **Runtime**
  - 0xffffffff
  - int f() {
    int x;
    ...
  }
  - malloc(sizeof(long));
  - static int x;
  - static const int y=10;

- **Known at compile time**
  - 0x00000000
We are going to focus on runtime attacks.

Stack and heap grow in opposite directions.
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

Stack pointer

push 1
push 2
push 3
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

0x00000000 0xffffffff

Heap Stack

Stack pointer

push 1
push 2
push 3
We are going to focus on runtime attacks.

Stack and heap grow in opposite directions.

Compiler provides instructions that adjusts the size of the stack at runtime.
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime

```
0x0000000000
0xffffffffffffffff
```

![Diagram showing stack and heap growth](image)
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime

---

**Diagram:**

- Stack and heap grow in opposite directions.
- Compiler provides instructions that adjust the size of the stack at runtime.
- Stack pointer.

---

**Push values:**
- push 1
- push 2
- push 3
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime

---

```
push 1
push 2
push 3
```
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

Stack pointer

push 1
push 2
push 3
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

```plaintext
push 1
push 2
push 3
return
```
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime

```
push 1
push 2
push 3
return
```
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

apportioned by the OS; managed in-process by \texttt{malloc}

+---+---+---+---+
|   | 3 | 2 | 1 |   
+---+---+---+---+
| Heap | Stack |
+---+---+

push 1
push 2
push 3
return
We are going to focus on runtime attacks.

Stack and heap grow in opposite directions:

- Compiler provides instructions that adjusts the size of the stack at runtime.

Heap:
- Apportioned by the OS.
- Managed in-process by `malloc`.

Stack:
- Pointer.

Focusing on the stack for now:
Stack layout when calling functions

• What do we do when we *call* a function?
  • What data need to be stored?
  • Where do they go?

• How do we *return* from a function?
  • What data need to be *restored*?
  • Where do they come from?

**Code examples**
(see `~/UMD/examples` in the VM)