Owing the Bits: Thinking about your Code from the Hackers Point of View

Elisa Heymann
Computer Architecture and Operating Systems Department
Universitat Autònoma de Barcelona
Elisa.Heymann@uab.es

Barton P. Miller
Computer Sciences Department
University of Wisconsin
bart@cs.wisc.edu

Condor Week
Madison May 2, 2013

This research funded in part by Department of Homeland Security grant FA8750-10-2-0030 (funded through AFRL). Past funding has been provided by NATO grant CLG 983049, National Science Foundation grant OCI-0844219, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.
What do we do

• **Assess Middleware:** Make cloud/grid software more secure

• **Train:** We teach tutorials for users, developers, sys admins, and managers

• **Research:** Make in-depth assessments more automated and improve quality of automated code analysis

## Our experience

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
<th>Vulnerabilities</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Condor</strong>, University of Wisconsin</td>
<td>Batch queuing workload management system</td>
<td>15</td>
<td>600 KLOC of C and C++</td>
</tr>
<tr>
<td><strong>SRB</strong>, SDSC</td>
<td>Storage Resource Broker - data grid</td>
<td>5</td>
<td>280 KLOC of C</td>
</tr>
<tr>
<td><strong>MyProxy</strong>, NCSA</td>
<td>Credential Management System</td>
<td>5</td>
<td>25 KLOC of C</td>
</tr>
<tr>
<td><strong>gIExec</strong>, Nikhef</td>
<td>Identity mapping service</td>
<td>5</td>
<td>48 KLOC of C</td>
</tr>
<tr>
<td><strong>Gratia Condor Probe</strong>, FNAL and Open Science Grid</td>
<td>Feeds Condor Usage into Gratia Accounting System</td>
<td>3</td>
<td>1.7 KLOC of Perl and Bash</td>
</tr>
<tr>
<td><strong>Condor Quill</strong>, University of Wisconsin</td>
<td>DBMS Storage of Condor Operational and Historical Data</td>
<td>6</td>
<td>7.9 KLOC of C and C++</td>
</tr>
</tbody>
</table>
Our experience

**Wireshark**, wireshark.org
Network Protocol Analyzer
2 vulnerabilities
2400 KLOC of C

**Condor Privilege Separation**, Univ. of Wisconsin
Restricted Identity Switching Module
2 vulnerabilities
21 KLOC of C and C++

**VOMS Admin**, INFN
Web management interface to VOMS data
4 vulnerabilities
35 KLOC of Java and PHP

**CrossBroker**, Universitat Autònoma de Barcelona
Resource Mgr for Parallel & Interactive Applications
4 vulnerabilities
97 KLOC of C++

**ARGUS 1.2**, HIP, INFN, NIKHEF, SWITCH
gLite Authorization Service
0 vulnerabilities
42 KLOC of Java and C
Our experience

**VOMS Core**  INFN
Virtual Organization Management System
1 vulnerability  161 KLOC of Bourne Shell, C++ and C

**iRODS, DICE**
Data-management System
9 vulnerabilities (and counting)  285 KLOC of C and C++

**Google Chrome**, Google
Web browser
1 vulnerability  2396 KLOC of C and C++

**WMS, INFN**
Workload Management System
in progress  728 KLOC of Bourne Shell, C++, C, Python, Java, and Perl
Learn to Think Like an Attacker
An Exploit through the Eyes of an Attacker

**Exploit:**
- A manipulation of a program’s internal state in a way not anticipated (or desired) by the programmer.

Start at the user’s entry point to the program: the *attack surface*:
- Network input buffer
- Field in a form
- Line in an input file
- Environment variable
- Program option
- Entry in a database
- ...

**Attack surface:** the set of points in the program’s interface that can be controlled by the user.
The Path of an Attack

```c
p = requesttable;
while (p != (struct table *)0) {
    if (p->entrytype == PEER_MEET) {
        found = (!strcmp (her, p->me) &&
                  !strcmp (me, p->her));
    } else if (p->entrytype == PUTSERVER) {
        found = !strcmp (her, p->me));
    }
    if (found)
        return (p);
    else
        p = p->next;
} return ((struct table *) 0);
```
Follow the *data and control flow* through the program, observing what state you can control:

- Control flow: what branching and calling paths are affected by the data originating at the attack surface?
- Data flow: what variables have all or part of their value determined by data originating at the attack surface?

Sometimes it’s a combination:

```java
if (inputbuffer[1] == 'a')
    val = 3;
else
    val = 25;
```

`val` is dependent on `inputbuffer[1]` even though it’s not directly assigned.
The Path of an Attack

```
... snprintf(buf, "/bin/mail %s", argv[i]) ...
... The Attack Surface

p = requesttable;
while (p != (struct table *)0) {
    if (p->entrytype == PEER_MEET)
        found = (!strcmp(her, p->me) &&
                    !strcmp(me, p->her));
    else if (p->entrytype == PUTSERVER)
        found = !(strcmp(her, p->me));
    if (found)
        return (p);
    else
        p = p->next;
} return ((struct table *)0);
```

The Impact Surface

```
... popen(buf, "w") ...
... The Impact Surface
```
An Exploit through the Eyes of an Attacker

The goal is to end up at points in the program where the attacker can override the intended purpose. These points are the *impact surface*:

- Unconstrained execution (e.g., exec’ing a shell)
- Privilege escalation
- Inappropriate access to a resource
- Acting as an imposter
- Forwarding an attack
- Revealing confidential information
- ...
The Path of an Attack

```c
... snprintf(buf, "/bin/mail %s", argv[i]) ...
... p = requesttable;
while (p != (struct table *)0)
{
    if (p->entrytype == PEER_MEET)
        {found = (!strcmp(buf, p->me) &&
                    !strcmp(me, p->her));}
    else if (p->entrytype == PUTSERVER)
        {found = !strcmp(buf, p->me));}
    if (found)
        return (p);
    else
        {p = p->next;}
} return ((struct table *) 0);
... popen(buf, "w") ...
... The Attack Surface

The Impact Surface
```
int foo()
{
    char buffer[100];
    int i, j;
    ...
    gets(buffer);
    ...
    jmp <evil addr>(buffer);
}
The stack smashing example is a simple and obvious one:

- The input directly modified the target internal state... ... no dependence on complex control or data flows.
- The attacker owned all the target bits, so had complete control over the destination address.
- No randomization
- No internal consistency checks
- No modern OS memory protection
- No timing issues or races
Evaluation: Finding Bits to Own

So, how do you find vulnerabilities in the face of these complexities?

– Complex flows:
  • *Taint analysis*: execute program in special simulation that tracks data from input buffers through execution, marking all the data and control-flow decisions affected by the data.
  • *Fuzz testing*: using unstructured or partially structured random input to try to crash the program.

  *Reliability is the foundation of security.*

– Randomness:
  • Repeated attempts: Sometimes patience is all that you need.
  • Grooming: A sequence of operations that bring the program to a known state, e.g.:
    – Cause a library to be loaded at a known address.
    – Cause the heap to start allocating at a known address.
    – Heap sprays: repeated patterns of code/data written to the heap so that at least one copy is in a useful place.
Prevention: Randomness

Create a moving target:

- Address space randomization (ASR): change the address of the code that contains the jump target from run to run.

In a classic stack smashing attack, the code was in the stack frame.

Also randomize addresses of code, heap, control blocks (e.g., Process Environment Block (PEB) on Windows), and mapped files.

- Stack layout randomization: several ways ...
  - Address of the start of the stack
  - Random padding between frames
  - Order of local variables and parameter layout
Prevention: Randomness

In practice, Linux:

• Support Address Space Layout Randomization (ALSR) since 2.6.12 (2005):
  – Stack: 19 bits of randomness on 16 byte boundaries.
  – Heap: 8 bits of randomness on page (often 4K) boundaries.
  – Code: Enabled by position independent executables (PIEs).

• Check the status of ALSR:
  
  ```
  cat /proc/sys/kernel/randomize_va_space
  ```

  One of the following values should be displayed:
  – 0: Disabled.
  – 1: (Conservative) Shared libraries and PIE binaries are randomized.
  – 2: (Full) Conservative settings plus randomize the start of \textit{brk} area.
Prevention: Randomness

In practice:

• Windows:
  – Available since Vista. Major improvements in Windows 7 and 8, especially for 64-bit executables. You sacrifice a lot of security with 32-bit executables.
  – Heap: Addition of heap guard pages, randomization of allocation order.
  – Code: Enabled by linking with /DYNAMICBASE
    • Better randomness for code appearing above 4GB in address space.
Prevention: Address Space Controls

Prevent code executing in data space:

- PAE (physical address extensions) on Intel (XD) or AMD (NX): prevent execution from certain pages, such as stack. Called data execution prevention (DEP) on Windows.

- Can do the same for heap variables, but would prevent JIT-based software, such as a Java virtual machine or binary profiler (e.g., Valgrind or Intel PIN)
Prevention: Consistency Checks

Stack canaries

– On function entry, when building stack frame, place a value on the stack, between the data and control information (typical, return address)
– The value is usual a random number that varies from run to run, even call to call.
– On function exit, check to see if canary value is still present.

– Turning on stack checking:
  • gcc: compile with \texttt{-fstack-protector-all}
  • Visual Studio: compile with \texttt{/GS} (on by default)
int foo() {
    char buffer[100];
    int i, j;
    \texttt{<push canary on stack>}
    ...
    gets(buffer);
    ...
    \texttt{<check canary value>}
    return(strlen(buffer));
}
Prevention: Consistency Checks

Heap consistency checks

– Store extra information about the size and layout of allocated and free memory regions in the heap.
– On each heap operation, e.g., malloc or free, and periodically other times, scan the heap for sensible structure.
– Can use tools like Valgrind, IBM Rational Purify, or Insure++ to check programs in a more detailed way for memory errors at runtime.

– Turning on heap checking:
  • gcc: compile with \texttt{-lmcheck} or call \texttt{mcheck} (or call \texttt{mprobe} for individual checks)
  • Windows: set heap check by running gflags.exe before running your program, or call \texttt{_heapchk} from within the program.
Would you like a tutorial taught at your site?

Tutorials for users, developers, administrators and managers:

– Security Risks
– Secure Programming
– Vulnerability Assessment

Contact us!

Barton P. Miller  Elisa Heymann
bart@cs.wisc.edu  Elisa.Heymann@uab.es
Questions?

http://www.cs.wisc.edu/mist