Owning 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

http://www.cs.wisc.edu/mist/papers/VAshort.pdf

Our experience

Condor, University of Wisconsin
Batch queuing workload management system
15 vulnerabilities — 600 KLOC of C and C++

SRB, SDSC
Storage Resource Broker - data grid
5 vulnerabilities — 280 KLOC of C

MyProxy, NCSA
Credential Management System
5 vulnerabilities — 25 KLOC of C

glExec, Nikhef
Identity mapping service
5 vulnerabilities — 48 KLOC of C

Gratia Condor Probe, FNAL and Open Science Grid
Feeds Condor Usage into Gratia Accounting System
3 vulnerabilities — 1.7 KLOC of Perl and Bash

Condor Quill, University of Wisconsin
DBMS Storage of Condor Operational and Historical Data
6 vulnerabilities — 7.9 KLOC of C and C++

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

THE UNIVERSITY of WISCONSIN MADISON

Universitat Autònoma de Barcelona

An Exploit through the Eyes of an Attacker

Exploit:

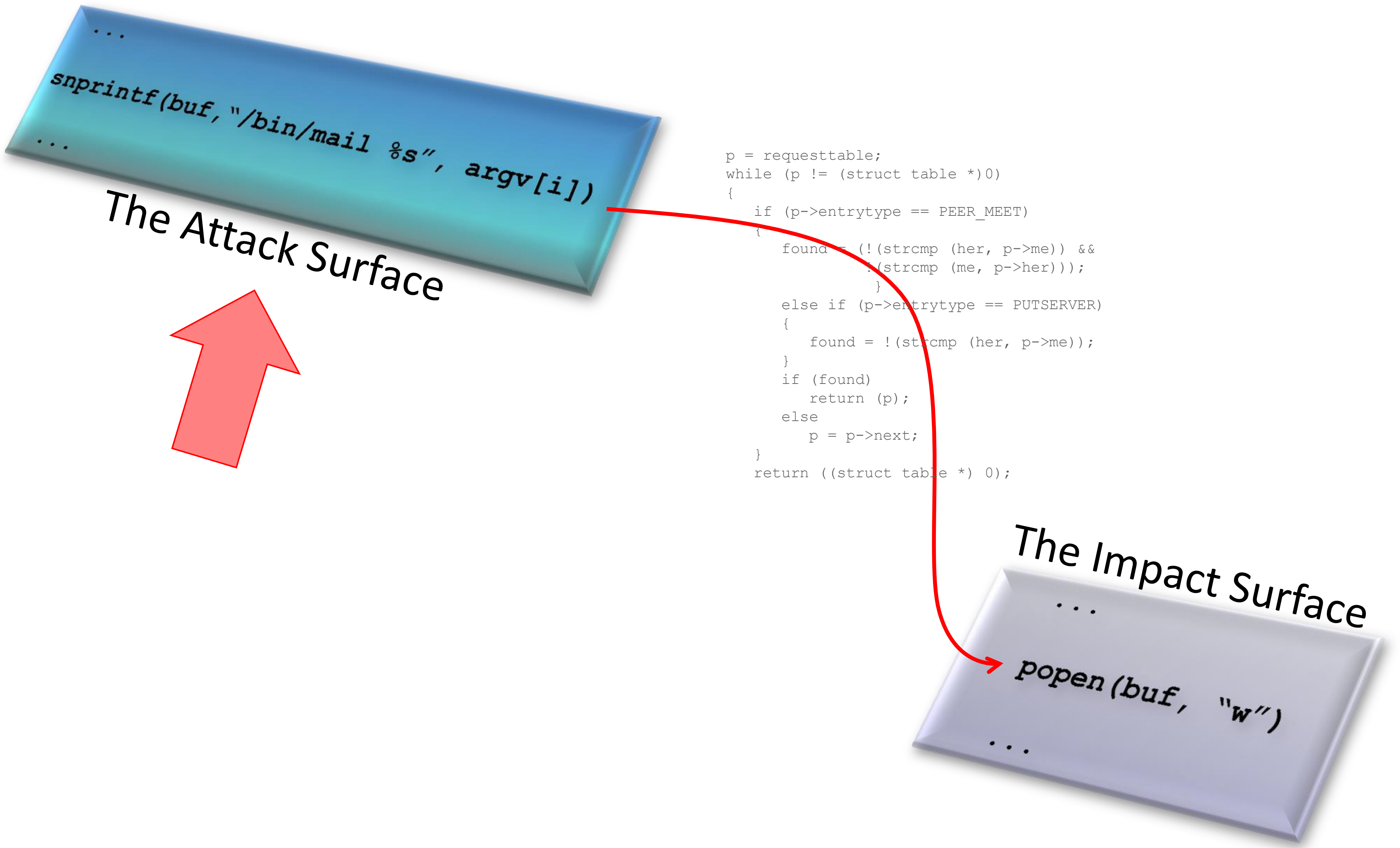
– A manipulation of a program's internal state in a way not anticipated (or desired) by the programmer.

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



An Exploit through the Eyes of an Attacker

Follow the *data and control flow* through the program, observing what state you can control:

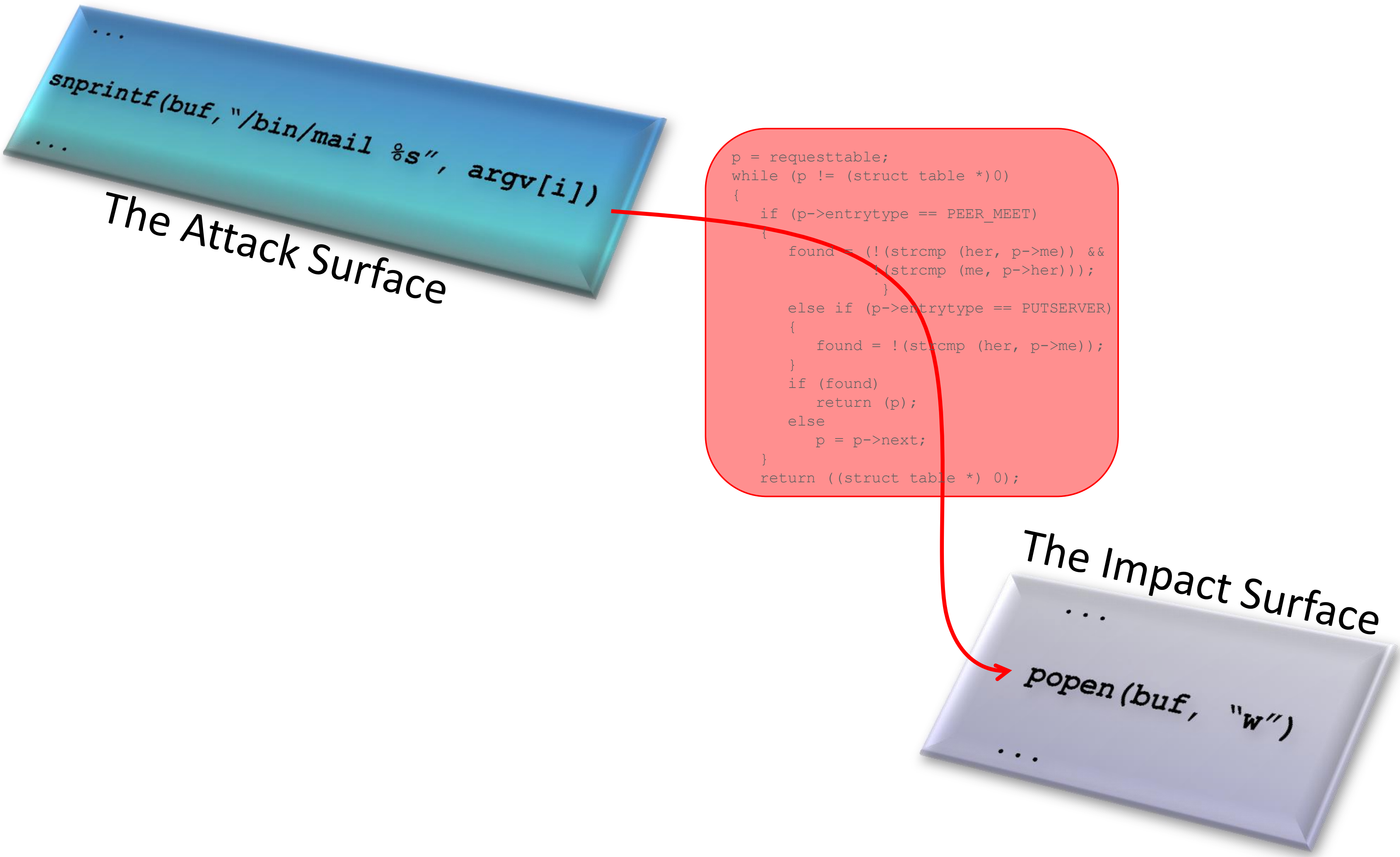
- Control flow: what branching and calling paths are affected by the data originating at the attack surface?
- Data flow: what variables have all or part of their value determined by data originating at the attack surface?

Sometimes it's a combination:

```
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

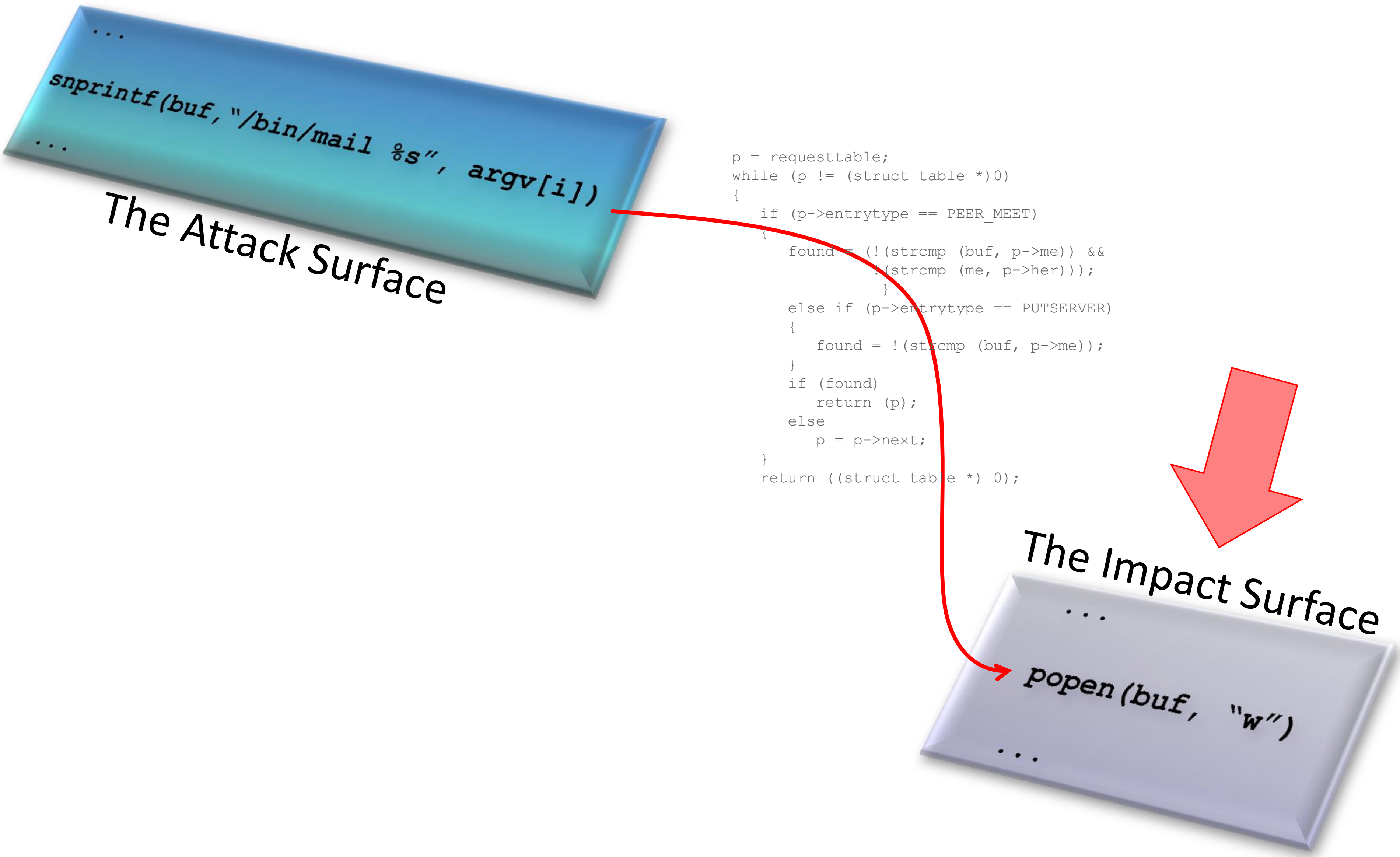


An Exploit through the Eyes of an Attacker

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

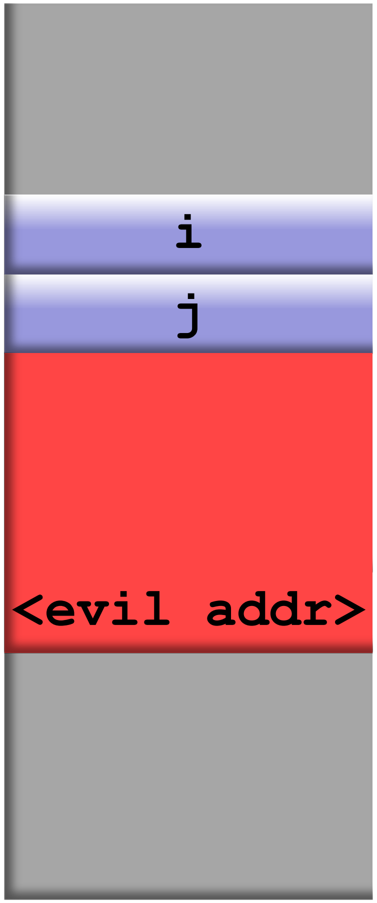


The Classic: A Stack Smash

```
int foo()
{
   char buffer[100];
   int i, j;
   …

   gets(buffer);

   …
   return(strlen(buffer));
   jmp <evil addr>
}
```



An Exploit through the Eyes of an Attacker

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 know 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 *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 `-fstack-protector-all`
  - Visual Studio: compile with `/GS` (on by default)

Prevention: Consistency Checks

```
int foo()
{
   char buffer[100];
   int i, j;
   <push canary on stack>
   …
   gets(buffer);
   …
   <check canary value>
   return(strlen(buffer));
}
```

THE UNIVERSITY of WISCONSIN MADISON

Universitat Autònoma de Barcelona

NATO OTAN

NSF

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 `-lmcheck` or call `mcheck` (or call `mprobe` for individual checks)
  - Windows: set heap check by running gflags.exe before running your program, or call `_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

bart@cs.wisc.edu

Elisa Heymann

Elisa.Heymann@uab.es

Questions?

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