Topics in Systems and Program Security

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

September 15, 2008
Presentations

• Structural Points
  ‣ Area
  ‣ Problem
  ‣ Why not solved (Related Work)
  ‣ Example
  ‣ Architecture
  ‣ Key Technologies (Can’t Do All)
  ‣ Evaluation
  ‣ Take Away

• These should be your own slides!!!
Presentations

• Similar to Summary
  • Same basic areas as a summary

• Different Than a Summary
  • Engage the audience
  • Identify an insight
  • Argue a point
  • Make an extension

• Relate to Security Concepts

• What Strikes You
Hints from Armando Fox

- Know Thy Jargon
- Keep the Big Picture in Mind
- Tell a Story
- Pace Yourself
- Tell ‘em What You Told ‘em
- Be Ready for Questions
Bouncer

- SOSP 2007
  - Systems conference
- Paper
  - Non-traditional SOSP paper
    - User-level program improvements
    - Many security conference references
    - Program language tools
- Importance of programming to systems
• Malicious inputs to programs
  ‣ Can lead to vulnerabilities…
  ‣ How do we detect and delete such inputs?
  ‣ How do we do it so that the service is not interrupted?
• Some solutions require killing the process
Example

```c
ProcessMessage(char* msg) {
    char buffer[1024];
    char p0 = 'A';
    char p1 = 0;

    if (msg[0] > 0)
        p0 = msg[0];

    if (msg[1] > 0)
        p1 = msg[1];

    if (msg[2] == 0x1) {
        sprintf(buffer, "\servers\%s\%c", msg+3, p0);
        StartServer(buffer, p1);
    }
}
```

Figure 1: Example vulnerable code: sprintf can overflow buffer.
Hasn’t This Been Done

• Well, yes, but…

• Problem 1: Detect attacks too late
  ‣ Example: CRED (Ruwase and Lam, NDSS 2004)
  ‣ Add bounds checks – restart on failure?
    • Why would that be?

• Problem 2: Techniques to detect attacks
  ‣ Static and dynamic detectors – have limitations
  ‣ But, generates filters automatically
    • Low overhead and no false positives
Approach

Figure 2: Bouncer architecture.
How’s It Work?

• Inputs
  ‣ Binary code and an exploit on that code
    • Log runtime attacks
    • DFI: if an instruction that wrote a memory location is not in the set determined by static analysis (error)

• Generate traces up to vulnerability point (VP)
  ‣ Nirvana generates traces

• Generate filter preconditions to reach VP
  ‣ Vigilante creates a symbolic representation of a filter
How’s It Work?

• Add **symbolic summaries** for library functions
  ‣ Remove/generalize some library-specific preconditions

• Use **precondition slicing** to remove unnecessary conditions
  ‣ Some dynamic and static analysis (main technology)

• Generate other exploits for vulnerability
  ‣ Further generalize based on repeating with other exploits

• Combine filter conditions found via these exploits
  ‣ No false positives
Symbolic Execution

• Generate Initial **Filter Condition**
  ‣ Trace Execution Path Generated by Exploit (Symbolically)

• Steps
  ‣ Replace input by symbolic version
    • Byte at index $i$ is $b_i$
  ‣ Execute instructions and symbolic values
    • Symbolic values are those that depend on some $b_i$
    • Generates a tree of opcodes with leaves as values
    • Define a total order on such instructions
Symbolic Execution

• Build a filter condition
  ‣ On every branch
    • If value tested is symbolic (depends on input)
    • Must compute taken/not-taken conditions depending on trace
  ‣ Also on indirect call or jump based on symbolic target

• Filter is a conjunction of these conditions
  ‣ No false positives (for that setting)
Detector Is Important

- Poor detection increases the length of traces
- Detect when observe the effect of an exploit
  - Rather than the vulnerability
  - Why would this effect accuracy?
- Traverse the trace backwards from the DFI error
  - Find the unsafe write
  - Rather than exploitable read
- Libraries are a challenge (DFI detects outside)
Precondition Slicing

- Technique to generalize the filters
- Extends notion of program slicing
Program Slicing

• Slicing model
  ▶ Identify the set of instructions that are relevant to a set of variables when a chosen instruction is reached
  • Set of instructions (the slice)
  • Set of variables (input variables)
  • Chosen instruction (vulnerability point)

• Why does this enable the removal of unnecessary filter conditions?
Classical Slicing Is Insufficient

- Static analysis for C/C++ includes excess instructions
  - Harder to tell if one instruction impacts another
- Dynamic slicing observes dynamic dependencies in actual traces
  - May miss some dependencies
- Precondition slicing combines advantages of static and dynamic, yet results in no false positives
Precondition Slicing

- Algorithm Overview
  - Inputs: Trace, Program Code, Alias Analysis
  - Trace
    - Entry for each instruction
    - Entries include instruction, mem addresses, and registers, symbolic and concrete values read

- Algorithm Maintains
  - Cur: trace entry
  - Slice: entries added to slice
  - Live: dependencies of instructions (and operands)
Slicing Algorithm

```c
ComputeSlice() {
    while (!trace.IsEmpty) {
        cur = trace.RemoveTail();
        if (cur.IsRet) {
            call = trace.FindCall(cur);
            if (MayWriteF(CalledFunc(call), live))
                Take(cur);
            else
                trace.RemoveRange(cur, call);
        } else if (cur.IsCall) {
            Take(cur);
            foreach (e in trace.CallArgSetup(cur)) {
                Take(e);
                trace.Remove(e);
            }
        } else if (cur.IsBranch) {
            if (!Postdominates(slice.head, cur)
                || WrittenBetween(cur, slice.head))
                Take(cur);
            else {
                if (MayWrite(cur, live))
                    Take(cur);
            }
        }
    }
}

void Take(cur) {
    slice.AddHead(cur);
    live.UpdateWritten(cur);
    live.AddRead(cur);
}
```

Figure 5: Pseudo-code for the slicing algorithm.
Slicing Algorithm

```
ComputeSlice() {
    while (!trace.IsEmpty) {
        cur = trace.RemoveTail();
        if (cur.IsRet) {
            call = trace.FindCall(cur);
            if (MayWriteF(CalledFunc(call), live))
                Take(cur);
            else
                trace.RemoveRange(cur, call);
        } else if (cur.IsCall) {
            Take(cur);
            foreach (e in trace.CallArgSetup(cur)) {
                Take(e);
                trace.Remove(e);
            }
        } else if (cur.IsBranch) {
            if (!Postdominates(slice.head, cur)
                || WrittenBetween(cur, slice.head))
                Take(cur);
        } else {
            if (MayWrite(cur, live))
                Take(cur);
        }
    }
}

void Take(cur) {
    slice.AddHead(cur);
    live.UpdateWritten(cur);
    live.AddRead(cur);
}
```

Figure 5: Pseudo-code for the slicing algorithm.
Slicing Algorithm

ComputeSlice() {
    while (!trace.IsEmpty) {
        cur = trace.RemoveTail();
        if (cur.IsRet) {
            call = trace.FindCall(cur);
            if (MayWriteF(CalledFunc(call), live))
                Take(cur);
            else
                trace.RemoveRange(cur, call);
        } else if (cur.IsCall) {
            Take(cur);
            foreach (e in trace.CallArgSetup(cur)) {
                Take(e);
                trace.Remove(e);
            }
        } else if (cur.IsBranch) {
            if (!Postdominates(slice.head, cur)
                || WrittenBetween(cur, slice.head))
                Take(cur);
        } else {
            if (MayWrite(cur, live))
                Take(cur);
        }
    }
}

void Take(cur) {
    slice.AddHead(cur);
    live.UpdateWritten(cur);
    live.AddRead(cur);
}

Figure 5: Pseudo-code for the slicing algorithm.
Slicing Algorithm

ComputeSlice() {
    while (!trace.IsEmpty) {
        cur = trace.RemoveTail();
        if (cur.IsRet) {
            call = trace.FindCall(cur);
            if (MayWriteF(CalledFunc(call), live))
                Take(cur);
            else
                trace.RemoveRange(cur, call);
        } else if (cur.IsCall) {
            Take(cur);
            foreach (e in trace.CallArgSetup(cur)) {
                Take(e);
                trace.Remove(e);
            }
        } else if (cur.IsBranch) {
            if (!Postdominates(slice.head, cur)
                || WrittenBetween(cur, slice.head))
                Take(cur);
        } else {
            if (MayWrite(cur, live))
                Take(cur);
        }
    }
}

void Take(cur) {
    slice.AddHead(cur);
    live.UpdateWritten(cur);
    live.AddRead(cur);
}

Figure 5: Pseudo-code for the slicing algorithm.

Normal: if may write to a value, take it
Slicing Invariant

• “Let F be the current filter that contains all the conditions in the initial filter that were added by instructions up to cur and the conditions that result from slice.”

• “Then all the execution paths obtained by processing inputs that match F execute the sequence of instructions in slice and the source operands of each of these instructions have equivalent concrete or symbol values across these paths.”

• Whew!
Impact of Dynamic Analysis

• Can remove unnecessary conditions (via entries) earlier than with static analysis

• Example:
  ‣ P0 and P1 have the same storage location in trace
  ‣ But this cannot be determined statically
  ‣ But can see this dynamically
    • I guess based on the invariant
  ‣ Then can remove preconditions based on P0
MayWrite

• Combine static and dynamic analysis
  ‣ To determine whether one instruction overwrites an operand

• Maintain a set of all operands (in instructions) that may alias a live operand (one from the vulnerability)
  ‣ MayWrite returns true if any of the operands written in that entry (cur) are in this set

  ‣ Invariant is maintained:
    • \( *p1 = msg[1] \) can be dropped because \( *p1 \) does not affect vulnerability
Evaluation

<table>
<thead>
<tr>
<th>service</th>
<th>false positives</th>
<th>false negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL server</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>ghttpd</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>stunnel</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 1: Accuracy of Bouncer filters.

Figure 8: Number of conditions in Bouncer filters after applying each technique. The number of conditions after symbolic execution is the number of conditions in Vigilante filters.

Figure 9: Total time to generate filters.

Figure 10: Number of iterations to generate filters.
Evaluation

In some deployment scenarios, it is easy to reduce filter generation times by exploiting parallelism. Since iterations in our filter generation process are independent, it can be parallelized by assigning each iteration to a different processor. For example, a large software vendor like Microsoft could run the filter generation process in a cluster with 1000 machines and then disseminate the filters to users of vulnerable software. This could speed up filter generation times by up to three orders of magnitude, for example, generating the filters for the SQL Server and stunnel vulnerabilities would take less than one minute.

In other scenarios, we can deploy a filter after the first iteration, which takes tens of seconds. Then we can deploy

Figure 11: Filter overhead for the Microsoft SQL server vulnerability as a function of message size.

Figure 12: Filter overhead for the ghttpd vulnerability as a function of message size.

Figure 13: Filter overhead for the nullhttpd vulnerability as a function of message size.
Take Away

• Defending against malicious inputs is a significant problem
  ‣ Prevents achievement of meaningful integrity

• Want to define filters to protect the program
  ‣ Automatically if at all possible

• Bouncer generates filters from exploit inputs and vulnerability point (detection)
  ‣ More accurately (based on static and dynamic analyses)

• Can we generate filters without exploits and vulnerabilities? Positive filters?