Who are we?

• Co-founder and Chief Scientist at Lastline, Inc.
  – Lastline offers protection against zero-day threats and advanced malware

• Professor in Computer Science at UC Santa Barbara
  – many systems security papers in academic conferences

• Member of Shellphish
Who are we?

- PhD Student at UC Santa Barbara
  - research focused primarily on binary security and embedded devices

- Member of Shellphish
  - team leader of Shellphish's CGC effort
What are we talking about?
The “Internet of Things”

Global Internet Device Installed Base Forecast

Source: Gartner, IDC, Strategy Analytics, Machina Research, company filings, BI estimates
Embedded software is everywhere
What is on embedded devices?

- Embedded Linux and user-space programs

- Custom OS and custom programs combined together in a *binary blob*
  - typically, the binary is all that you get
  - and, sometimes, it is not easy to get this off the device
Binary Analysis

Binary analysis

noun | bi·na·ry anal·y·sis | \ˈbī-nə-rē ə-ˈna-lə-səs\

1. The process of automatically deriving properties about the behavior of binary programs
2. Including static binary analysis and dynamic binary analysis
Goals of Binary Analysis

- Program verification
- Program testing
- Vulnerability excavation
- Vulnerability signature generation
- Reverse engineering
- Vulnerability excavation
- Exploit generation
Static Binary Analysis

– reason over multiple (all) execution paths
– can achieve excellent coverage
– precision versus scalability trade-off
  • very precise analysis can be slow and not scalable
  • too much approximation leads to wrong results (false positives)
– often works on abstract program model
  • for example, binary code is lifted to an intermediate representation
Dynamic Binary Analysis

- examine individual program paths
- very precise
- coverage is (very) limited
- sometimes hard to properly run program
  - hard to attach debugger to embedded system
  - when code is extracted and emulated, what happens with calls to peripherals?
Challenges of Static Binary Analysis

- Get the binary code

- Binaries lack significant information present in source

- Often no clear library or operating system abstractions
  - where to start the analysis from?
  - hard to handle environment interactions
From Source to Binary Code

compile → link → strip

EXE
From Source to Binary Code

compile

link

strip

type info
function names
variable names
jump targets
(Linux) system call interface is great
  – you know what the I/O routines are
    • important to understand what user can influence
  – you have typed parameters and return values
  – let’s the analysis focus on (much smaller) main program

OS is not there or embedded in binary blob
  – heuristics to find I/O routines
  – open challenge to find mostly independent components
Missing OS and Library Abstractions

• Library functions are great
  – you know what they do and can write a “function summary”
  – you have typed parameters and return values
  – let’s the analysis focus on (much smaller) main program

• Library functions are embedded (like static linking)
  – need heuristics to rediscover library functions
  – IDA FLIRT (Fast Library Identification and Recognition Technology)
  – more robustness based on looking for control flow similarity
Types of Vulnerabilities

• Memory safety vulnerabilities
  – buffer overrun
  – out of bounds reads (heartbleed)
  – write-what-where

• Authentication bypass (backdoors)

• Actuator control!
Authentication Bypass

Prompt

Authentication

Success

Failure
Authentication Bypass

Backdoor e.g. strcmp()
Authentication Bypass

Prompt

Authentication

Success

Failure

Backdoor e.g. strcmp()

Hard to find.
Modeling Authentication Bypass

Prompt

Authentication

Success

Failure

Backdoor e.g. `strcmp()`

Hard to find.

Easier to find!
Input Determinism

Can we determine the input needed to reach the success function, just by analyzing the code?

The answer is NO
Input Determinism

Can we determine the input needed to reach the success function, just by analyzing the code?

The answer is YES
Modeling Authentication Bypass

Prompt

Authentication

Success

Failure

Backdoor e.g. strcmp()

Easier to find!

But how?
Finding “Authenticated Point”

- **Without OS/ABI information:**
  - Manual reverse engineering
  - Program outputs/references certain strings (like “welcome admin”)
  - Program accesses sensitive memory regions

- **With ABI information:**
  - Program calls sensitive syscalls
  - Program accesses sensitive resources/files
Using angr to Hunt for Vulnerabilities

- Program
- Security policies

- Static Analysis
- Symbolic Execution
- Security Policy Checker

POCs
angr: A Binary Analysis Framework

- Binary Loader
- Static Analysis Routines
- Symbolic Execution Engine
- Control-Flow Graph
- Data-Flow Analysis
- Value-Set Analysis
angr: A Binary Analysis Framework

- Binary Loader
- Static Analysis Routines
- Symbolic Execution Engine
- Control-Flow Graph
- Data-Flow Analysis
- Value-Set Analysis
"How do I trigger path X or condition Y?"

- Dynamic analysis
  - Input A? No. Input B? No. Input C? ...
  - Based on concrete inputs to application.
- (Concrete) static analysis
  - "You can't"/"You might be able to"
  - Based on various static techniques.

We need something slightly different.
"How do I trigger path X or condition Y?"

1. Interpret the application.
2. Track "constraints" on variables.
3. When the required condition is triggered, "concretize" to obtain a possible input.
Constraint solving:

- Conversion from set of constraints to set of concrete values that satisfy them.
- NP-complete, in general.
Symbolic Execution

```python
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"

<table>
<thead>
<tr>
<th>State A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
</tr>
</tbody>
</table>
| x = ???

<table>
<thead>
<tr>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
</tr>
</tbody>
</table>

Symbolic Execution
```python
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```
```python
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```

<table>
<thead>
<tr>
<th>State AA</th>
<th>State AB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Variables</td>
</tr>
<tr>
<td>x = ???</td>
<td>x = ???</td>
</tr>
<tr>
<td>Constraints</td>
<td>Constraints</td>
</tr>
<tr>
<td>x &lt; 10</td>
<td>x &gt;= 10</td>
</tr>
</tbody>
</table>

x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"

<table>
<thead>
<tr>
<th>State ABA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
</tr>
<tr>
<td>x = ???</td>
</tr>
<tr>
<td>Constraints</td>
</tr>
<tr>
<td>x &gt;= 10</td>
</tr>
<tr>
<td>x &lt; 100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concretized ABA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
</tr>
<tr>
<td>x = 99</td>
</tr>
</tbody>
</table>
# Symbolic Execution - Pros and Cons

## Pros

- Precise
- No false positives (with correct environment model)
- Produces directly-actionable inputs

## Cons

- Not scalable
  - Constraint solving is NP-complete
  - Path explosion
Path Explosion

∞ paths to here

5 paths to here
an gr: A Binary Analysis Framework

- Binary Loader
- Static Analysis Routines
- Symbolic Execution Engine
- Control-Flow Graph
- Data-Flow Analysis
- Value-Set Analysis
angr: A Binary Analysis Framework

- Binary Loader
- Static Analysis Routines
- Symbolic Execution Engine
- Control-Flow Graph
- Data-Flow Analysis
- Value-Set Analysis
angr: A Binary Analysis Framework

- Binary Loader
- Static Analysis Routines
- Symbolic Execution Engine
- Control-Flow Graph
- Data-Flow Analysis
- Value-Set Analysis
angr: A Binary Analysis Framework

- Binary Loader
- Static Analysis Routines
- Symbolic Execution Engine
- Control-Flow Graph
- Data-Flow Analysis
- Value-Set Analysis
## Value Set Analysis

<table>
<thead>
<tr>
<th>Memory access checks</th>
<th>Type inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable recovery</td>
<td>Range recovery</td>
</tr>
<tr>
<td>Wrapped-interval analysis</td>
<td></td>
</tr>
<tr>
<td>Value-set analysis</td>
<td></td>
</tr>
<tr>
<td>Abstract interpretation</td>
<td></td>
</tr>
</tbody>
</table>
Value Set Analysis

```plaintext
{  
  (global, (4[0x601000, 0x602000], 32)),  
  (stack_0x400957, (8[-0xc, -0x4], 32))  
}
```

<table>
<thead>
<tr>
<th></th>
<th>global</th>
<th>stack_0x400957</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x601000, 0x601004</td>
<td>0x601000, 0x601004</td>
</tr>
<tr>
<td></td>
<td>0x601008, 0x60100c</td>
<td>- 0xc</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>- 0x4</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
What have we used this for?

- Binary Loader
- Static Analysis Routines
- Symbolic Execution Engine
- Control-Flow Graph
- Data-Flow Analysis
- Value-Set Analysis
The Cyber Grand Challenge!

CB
vulnerable program

Cyber Reasoning System

POV
exploit

RB
patched program
The Shellphish CRS

PCAP

Autonomous processing

Autonomous vulnerability scanning

CB

Autonomous patching

Test cases

Proposed POVs

Proposed RBs

Autonomous service resiliency

POV

RB
The Shellphish CRS

PCAP

Autonomous processing

Autonomous vulnerability scanning

Autonomous patching

Test cases

Proposed POVs

Proposed RBs

Autonomous service resiliency

POV

RB