Miniaturization

“Why, sometimes I've believed as many as six impossible things before breakfast.”
— Lewis Carroll, Alice in Wonderland

Jason Larsen
Blackhat 2014
Who Am I?

- Jason Larsen
- CyberSecurity Researcher specializing in critical infrastructure
A play presentation in two parts

- I submitted two talks to Black Hat and ..... they said to do both of them at the same time
- Creating a kick@#$% SCADA attack firmware modification in two acts
  - Act I: Making the attack code really small
  - Act II: Efficiently inserting the rootkit into the firmware
- Popcorn Warning
  Lots of algorithms and assembly code ahead
Could You Hide an Entire Attack in a Pressure Meter?

- Small microcontroller
  - Kilobytes of memory (total)
  - Very little CPU power
  - Kilobytes of flash (total)
Eleven Years Ago
(And yes, it was lame even then)
Record and Playback

- The operator’s screens didn’t update in this video
- It was created using the trusty record-and-playback method
- What if we want to go small?
- What if we want to go really small?
- What if we want to go down into the sensors?
The Scenario

Valve

Water Flow

Shock Wave

Reflected Shock Wave

Physical Movement

Valve Closes

Shockwave

Reflected Wave
The Scenario

- The shockwave travels at the speed of sound in water
- Or, if pipe is elastic
- The optimal interval to cycle the valve
  - $X$ is the time between valve closing
  - $Y$ is the time between the pressure wave and the rarefaction wave

\[
t = \frac{L}{A}
\]

\[
E = \frac{E_{\text{water}} \times T_{\text{pipe}} \times E_{\text{pipe}}}{T_{\text{pipe}} \times E_{\text{pipe}} + D_{\text{pipe}} \times E_{\text{water}}}
\]

\[
\frac{2X + Y}{4}
\]

*Fluid Dynamics. Professional Publications Inc.*
Supersampling

*Mechanical Vibration and Shock Measurements
Act I – Making the Attack Code Really Small

Popcorn Alert! Lots of assembly ahead
Miniaturizing Firmware Attack Code

• Miniaturizing the Attack Code
  – *Spoofing with Runs Analysis*
    – Triangles for Filtering Noise
    – Scale-free matching for Watching the Process

• Inserting the Attack Code into the Firmware
  – MicroOps
  – Binary Normal Form
  – Abusing Needleman Wuncsh to Merge Firmware
  – Metasploit for Firmware

• Demos
Sensor Noise
(This isn’t going to fool anyone)

Anyone looking at this will think “dead sensor”
The forensics team will zoom on this immediately
Sensor Noise

- Humans are really good at spotting differences in "randomness"
- Even on graphical displays, operators get used to the "jiggle" in the visualization
Sensor Noise
A Random Walk

- Just adding randomness
  - It’s easy for a human to spot where the spoof starts
  - This doesn’t preserve the “spikiness”, “width”, and “gaps” of the original
Sensor Noise

- If you’re a math major, you’re probably shouting “Yeah! FFT!”

Total Flash

This won’t fit

Your Favorite FFT Library
Scaling and Shifting

Scaling can increase magnitude of the noise.

Shifting requires an averaging function to eliminate stair steps during adjustments.

These are solvable problems but they grow bigger as you try to get it right.
Runs Analysis

• Most of these techniques require that the attacker have access to previously recorded data to get the algorithm right.
  – What if we don’t get to see the sensor noise before we start?
• Runs analysis can spoof the sensor noise with no preknowledge of the data.
• Sensor noise can be treated as a random walk
• Random walks can be characterized through an analysis of the length and frequency of runs
Runs Analysis

- During a learning phase, count the runs

390.3
390.4  +3 increasing (moved 0.3)
390.6  -1 decreasing (moved -0.3)
390.3  -1 decreasing (moved -0.3)
390.5  +3 increasing (moved 0.6)
390.9
391.1  -1 decreasing (moved -0.8)
391.2
390.9
390.9
390.8

As expected this gives a nice normal distribution
Runs Analysis

- Taking the average movement of a runs bucket turns into a slope and a length

Chaining line segments together reproduces the noise

+5  +4  +3  -1  -2
Runs Analysis

- The playback algorithm is really simple
  - Add up all the positive/negative buckets
  - Choose a random number \(0 < x < \text{sum(buckets)}\)
  - Move by average bucket value for bucket samples
  - If desired is above current, choose from positive buckets otherwise choose from negative buckets
Runs Analysis

We get nice, believable sensor noise with no prior knowledge of the system.
Runs Analysis

- We have to fit this on the microcontroller. How big is the code+data?
  - Just over 400 bytes depending on linker constraints
  - ARM, X86, and PPC are similar in size
- We can definitely fit that inside a pressure sensor
• Miniaturizing the Attack Code
  – Spoofing with Runs Analysis
  – **Triangles for filtering noise**
  – Scale-free Matching for Watching the Process

• Inserting the Attack Code into the Firmware
  – MicroOps
  – Binary Normal Form
  – Abusing Needleman Wunsch to Merge Firmware
  – Metasploit for Firmware

• Demos
Leveling

• We’re going to be attacking the process and making changes
• We need to preserve the small changes that are expected so the forensics guys can match them up later
• We need to remove the big changes so the logs don’t show what we’ve been doing
We need to transfer this artifact

But not this trend
Leveling

- How big is an artifact?
- How big is a disturbance?
- Do I need a different algorithm for every type of signal?
- What if I don’t get to see the signal beforehand to choose my algorithm?
Leveling
Moving Averages

• Everyone starts with a moving average to filter out the data from the noise
  – This might not be the best approach
• Even simple algorithms can be large when the size of the data is taken into account
Moving Average 2 point

Where is the peak?
Will this detect 50 peaks?

4 Bytes to store the data
Moving Average 50 point

These will still get in the way

200 Bytes to store the data
Moving Average 200 point

Looking good, but at what cost?

800 Bytes to store the data
Peaks are detected 100 samples late.
Beyond Moving Averages
Fitting Curves to the Data

• A moving average is an example of a scale-dependent algorithm
• How many points should be applied to smooth out the curve?
  – It’s impossible to know without an example of the data
• LOTS of code is needed to deal with scaling factors
  – Mm/Hg, cm/h20, Pascals?
  – More than all the rest of the attack code combined
Forget sine waves. Your trig library isn’t going to fit either

Don’t forget all those nasty sensor glitches
Triangles

- Triangles are a good-enough approximation of the process data
- We just need a very small algorithm to fit triangles onto the process data
- How big is the optimal triangle?
  - The largest features are the ones you care about
  - We need an algorithm that will produce triangles that is scale independent
  - The triangles should all cover a similar area
Think of the process data as a set of triangles. Triangles are cheap and easy.
We can make some assumptions about the data

- The process is not running out of control therefore, it will oscillate as the feedback mechanisms control the process
- Artifacts smaller than the noise are too small to affect the process
- There isn’t significant hysteresis in the system
Triangles
(Still tweaking this one)

1. A simple algorithm
2. Declare a vertex at the first value
3. Choose an arbitrary starting window $n$. Calculate or estimate a smoothing factor $s=\log(n)$.
4. Note the minimum and the maximum values in the window.
5. Draw a triangle from the origin through the minimum and maximum values and ending in a vertical line at $n$.
6. Declare a vertex at the midpoint of the vertical line at $n$.
7. Start drawing a second triangle from the vertex using the slopes of the previous triangle.
8. Count $y,z$ samples that are above/below the triangle.
9. When $y$ or $z > s$, declare a vertex at the midpoint of the vertical line through the current sample.
10. If $y<z$, increase the slope of the top and decrease the slope of the bottom line otherwise do the opposite.
11. If the number of samples between the current sample and the last vertex $< 4n$, then increase $n$.
12. If at any time there has been no vertex in $4n$ samples, declare a vertex at the midpoint of the line through the current sample and decrease $n$.
13. Go to step 6
Transferring Artifacts

• Now that the triangles are complete
  – Declare that the midpoint of each line segment should be scaled to the spoof value
  – The difference from the line segment to the observed data is averaged into the spoof data
Scaling and Leveling
• Miniaturizing the Attack Code
  – Spoofing with Runs Analysis
  – Triangles for Filtering Noise
  – **Scale-free Matching for Watching the Process**
• Inserting the Attack Code into the Firmware
  – MicroOps
  – Binary Normal Form
  – Abusing Needleman Wuncsh to Merge Firmware
  – Metasploit for Firmware
• Demos
Artifact Extraction

• We need to spot the pressure wave and the reflected wave
• We can extract the state of the process using the triangles
• This saves CPU time because we’re only running this logic when we declare a new vertex
Artifact Extraction

Something happened
-Slopes changed
-Lengths changed
Artifact Extraction

- For our attack model we only need two artifacts
  - When did the pressure wave hit?
  - When did the reflected wave hit?

\[
\frac{2X + Y}{4}
\]
Scale Free Description

Modeling says it should look like this

We need to cycle the valve at:
X = (0.55 - 0.4)
Y = (0.65 - 0.55)
2X + Y = 0.4 seconds

It actually looks like this
Line Segments

Triangle Strips

Scale Free Description - Ratio of areas of adjacent triangles

.31-.33
.29-.33
.21-.29

Ratio of areas between adjacent triangles (I could have also used ratios of the angles)
Triangles

- How big is the triangle algorithm? We have to fit it into a pressure sensor, after all.
  - Approx 700 bytes (Ouch!)
Total Size

- Sensor Noise ~ 400 bytes
- Triangles ~ 700 bytes
- DNP CRC – 272 bytes (ouch!)
- Protocol and Glue Logic ~ 600 bytes

- Total Payload – 2174 bytes
  - That’s about 0.7% of the total flash
Act II – Inserting the Code into the Firmware

Popcorn Alert! Lots of assembly ahead
Inserting the Rootkit into the Firmware

- I still need to make my payload smaller
  - To make it smaller, I need to reuse the existing code.
- Debugging
  - If I’m reusing existing code, how do I debug it?
  - What if the existing code has side effects?
- Portability
  - I don’t want to recode my rootkit for every single sensor I want to invade.
Parallel Construction

- I’m going to write and debug my attack code on my MacBook (X86), debug it, and then deploy it on an pressure sensor (MSP430).
- I need to be able to translate between those two different architectures.
Example Code

```c
int CalcSomething(int x){
    int total = 0;
    int i;
    for (i=0;i<x;i++){
        total=total+i;
    }
    return total;
}
```
MSP430 Assembly

```
.def CalcSomething
CalcSomething:
push.w R4  
mov.w SP, R4  
incl.w R4  
add.w #0FFFAh, SP  
mov.w R15, OFFFC(R4)  
clr.w OFFF8h(R4)  
clr.w OFFFAh(R4)  
jmp loc_22

loc_22:
cmp.w OFFFC(R4), OFFFAh(R4)  
jl loc_18

loc_18:
add.w OFFFAh(R4), OFFF8h(R4)  
inc.w OFFFAh(R4)

mov.w OFFF8h(R4), R15  
add.w #6, SP  
pop R4  
ret
; End of function CalcSomething
```
ARM Assembly

MOV    R3, #0
STR    R3, [R11,#var_8]
MOV    R3, #0
STR    R3, [R11,#var_C]
B      loc_40

loc_40
LDR    R2, [R11,#var_C]
LDR    R3, [R11,#var_10]
CMP    R2, R3
BLT    loc_24

loc_24
LDR    R2, [R11,#var_8]
LDR    R3, [R11,#var_C]
ADD    R3, R2, R3
STR    R3, [R11,#var_8]
LDR    R3, [R11,#var_C]
ADD    R3, R3, #1
STR    R3, [R11,#var_C]

LDR    R3, [R11,#var_8]
MOV    R0, R3
SUB    SP, R11, #0
LDR    R11, [SP+var_s0], #4
BX     LR

; End of function CalcSomething
; .text ends
Are they different?

- We can’t directly compare the two assemblies
• Miniaturizing the Attack Code
  – Spoofing with Runs Analysis
  – Triangles for Filtering Noise
  – Scale-free Matching for Watching the Process
• Inserting the Attack Code into the Firmware
  – **MicroOps**
    – Binary Normal Form
    – Abusing Needleman Wuncsh to Merge Firmware
    – Metasploit for Firmware
• Demos
MicroOps

- Assembly language operations are actually complex
  - They can be described using several smaller operations
- Push EAX is actually complex instruction with two operations
  - Subtract 4 from the stack pointer
  - Move EAX into the memory pointed to by the stack pointer

\[
\text{PUSH EAX} \quad \Rightarrow \quad \text{ESP} := \text{ESP}-4 \quad [\text{ESP}] := \text{EAX}
\]
Let’s break these two down into MicroOps

Apples->Pears
Oranges->Pears

Now They are the same language!

But….Not exactly the same yet
• Miniaturizing the Attack Code
  – Spoofing with Runs Analysis
  – Triangles for Filtering Noise
  – Scale-free Matching for Watching the Process
• Inserting the Attack Code into the Firmware
  – MicroOps
  – **Binary Normal Form**
  – Abusing Needleman Wuncsh to Merge Firmware
  – Metasploit for Firmware
• Demos
What we need is a set of rules. Tame the chaos.
I call this set of rules Binary Normal Form
We apply all the rules, we have a good chance of converting the structure of the two MicroOp trees into the same tree.
Binary Normal Form

1. All loads and stores are via a register.
2. All branches are positive form “Jump if Equal” instead of “Jump if not Equal”.
3. The true branch always comes first (Jump to false).
4. ……
Binary Normal Form

R3:=0
[R11+8]:=R3
[R11+C]:=R3
PC:=loc_40
R3:=[R11+8]
R2:=[R11+C]
R3:=R2+R3
[R11+8]:=R3
R3:=[R11+C]
R3:=R3+1
[R11+C]:=R3
R2:=[R11+C]
R3:=[R11+10]
IF R2< R3 THEN PC:=loc_24

TMP1:=0
[R4+8]:=TMP1
[R4+10]:=TMP1
PC:=loc_22
TMP1:=[R4+8]
TMP2:=[R4+10]
TMP3:=TMP1+TMP2
[R4+8]:=TMP3
[R4+10]:=TMP3
TMP1:=[R4+10]
TMP1:=TMP1+1
[R4+10]:=TMP1
R2:=TMP1
R3:=[R11+10]
IF TMP1<TMP2 THEN PC:=loc_18

Excellent! They kinda match!!

It’s not an exact match. They use different registers and different stack offsets. Compilers may have ordered things differently.
Infinite Register File

- What can we do to normalize the registers and stack variables?
- It would be a shame we couldn’t compare two chunks of code simply because the compiler chose a different register.
- If there were an infinite number of registers, a compiler would never need to reuse them.
  - There would also be no need for stack variables.
Infinite Register File

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR</td>
<td>R11, [SP,#-4+var_s0]!</td>
<td>Allocate 5 Registers</td>
</tr>
<tr>
<td>ADD</td>
<td>R11, SP, #0</td>
<td>Allocate 1 Register for the new base pointer</td>
</tr>
<tr>
<td>SUB</td>
<td>SP, SP, #0x14</td>
<td>Becomes move stack into base</td>
</tr>
<tr>
<td>STR</td>
<td>R0, [R11,#var_10]</td>
<td>Allocate 5 Registers</td>
</tr>
<tr>
<td>MOV</td>
<td>R3, #0</td>
<td>Becomes move zero into a register</td>
</tr>
<tr>
<td>STR</td>
<td>R3, [R11,#var_8]</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>R3, #0</td>
<td></td>
</tr>
<tr>
<td>STR</td>
<td>R3, [R11,#var_C]</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>loc_40</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>R2, [R11,#var_8]</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>R3, [R11,#var_C]</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>R3, R2, R3</td>
<td></td>
</tr>
</tbody>
</table>
Infinite Register File

- Nasty stack operations are eliminated
- The two code segments match!
- We can say that they are the same logic (minus the register width).
• Miniaturizing the Attack Code
  – Spoofing with Runs Analysis
  – Triangles for Filtering Noise
  – Scale-free Matching for Watching the Process
• Inserting the Attack Code into the Firmware
  – MicroOps
  – Binary Normal Form
  – **Abusing Needleman Wuncsh to Merge Firm**
  – Metasploit for Firmware
• Demos
Modified Code

int CalcSomething(int x){
    int total = 0;
    int i;

    for (i=0;i<x;i++){
        total=total+i;
    }
    return total;
}

int EvilSomething(int x){
    int total = 0;
    int i;

    for (i=0;i<x;i++){
        total=total+i+4;
    }
    return total;
}

What if I made some changes?
Edit Distance

• How close are these two functions?
• One way to measure that is the edit distance
  • How many IF statements would it take to make them the same?
### Edit Distance

These two functions differ with an edit distance of 1.
Edit Distance

• That was a trivial example
• How can we find the edit distance between two pieces of code in a more generic way?
• We can steal from the biologists and use protein matching algorithms
  – Needleman-Wunsch can be used to find the edit distance between two strings
• We can adapt that for our uses
Needleman Wunsch

Inserting Code is Fun
Inserting Rootkits is Fun

Inserting _Co___de is Fun
Inserting Rootkits is Fun
Needleman Wunsch

Inserting Code is Fun
Inserting Rootkits is Fun

The strings have an edit distance of 2

18 Characters the same - 10 characters different
Edit Distances Between Functions

S1:=0
S2:=0
PC:=PC+2
S1:=S1+S2
S2:=S2+1
IF S2< ARG1 THEN PC:=PC-2

S1:=0
S2:=0
PC:=PC+3
S3:=S2+4
S1:=S1+S3
S2:=S2+1
IF S2<ARG1 THEN PC:=PC-3

• What if we turned these MicroOps into letters?
• We could calculate the edit distance between any two functions
• It would even tell us where to put the IF statements
Edit Distances Between Functions

S1 := 0
S2 := 0
PC := PC + 2
S1 := S1 + S2
S2 := S2 + 1
IF S2 < ARG1 THEN PC := PC - 2

S1 := 0
S2 := 0
PC := PC + 2
S1 := S1 + S2
S2 := S2 + 1
IF S2 < ARG1 THEN PC := PC - 2
Edit Distances Between Functions

The string shows where to add the IF statements to make the functions that same.
Edit Distances

If we know how costly an IF statement is in the target architecture, we can figure out if merging these two function will save space in the final firmware.
Matching Call Trees

• Now that we can match two functions, why not something bigger?
• We can take each of our leaf functions and see if the parents of that leaf function also match.
Matching Call Trees

After matching, it is now possible to calculate the edit distance of an entire subsystem

Distance: 8
Finally!
Inserting the Rootkit into the Firmware!

• For each function in the rootkit, I have found a best match function in
  the target firmware
  • If mine has a CRC-16 and the target has a CRC-16, they will have a small edit distance and get merged together
  • Any orphans that simply don’t match will need to be added to the end
• I can even merge two functions in the target together to gain even more space
• Now simply reverse the process from BNF back to the target assembly
• Instant firmware rootkit!

```c
uint16_t crc16_update(uint16_t crc, uint8_t a){
    int i;
    crc ^= a;
    for (i = 0; i < 8; ++i){
        if (crc & 1)
            crc = (crc >> 1) ^ 0xA001;
        else
            crc = (crc >> 1);
    }
    return crc;
}
```
• Miniaturizing the Attack Code
  – Spoofing with Runs Analysis
  – Triangles for Filtering Noise
  – Scale-Free Matching for Watching the Process
• Inserting the Attack Code into the Firmware
  – MicroOps
  – Binary Normal Form
  – Abusing Needleman Wuncsh to Merge Firmware
    – Metasploit for Firmware
• Demos
Random Code from the Internet

• Nobody would ever just copy code from the Internet would they?
• Since we can compare code, we can search to see if this code is in that firmware
Future: Metasploit for Firmware

- There are common pieces of software used throughout industrial control systems.
  - i.e. SquareD DNP stack
- As long as our rootkit only needs functionality from the common piece of code, the merge will be self-contained.
  - It can be inserted automatically without a human
  - No need to understand the CPU
  - No need to deal with the version differences
Demos
Questions

- Jason Larsen
- IOActive, Inc.