Miniaturization

Jason Larsen
CyberSecurity Researcher

Black Hat Las Vegas, 2014

Abstract

Too often researchers ignore the actual process in SCADA research. “I hacked into the control system so I win!” Ignoring the process means missing some of the interesting problems. Miniaturization is one of those interesting problems. How small can an attacker make SCADA attack code? Can it be made small enough to fit on a pressure sensor? This is an interesting problem.

This paper covers a number of techniques for miniaturizing SCADA attack code down to its smallest size and then efficiently inserting it into the firmware of a target device. The paper is very algorithm and assembly heavy and is targeted towards and audience looking to research small attack code or researchers looking to do forensics of advanced payloads.

The paper is broken into two parts. The first part covers some novel algorithms designed to perform the actual attack logic with a very small amount of data and code. The second part covers a method for efficiently merging the attack code with the existing code in the firmware while making maximum reuse of the existing code already in the firmware. It introduces an intermediate language called Binary Normal Form that can be used to translate assembly instructions from one architecture to another.

These techniques show that it may be possible to create a Metasploit-like framework that can insert off-the-self attack code into a wide variety of firmware with minimal human interaction.

IOActive™

Comprehensive Information Security

The Scenario

The techniques described in a surprising number of papers on SCADA hacking are difficult to apply to real-world scenarios. They assume unreasonable access or take other shortcuts. To help avoid this pitfall, this paper uses a very specific scenario as a backdrop for the examples.

The attack code in this paper will attempt destroy a piping system by setting up a standing wave of pressure in a pipe. This setup was actually built in a lab, but the results of the physical damage experimentation will appear in another paper. This paper will describe only the computer science aspects of the experiment and not the results of the physics experiment.

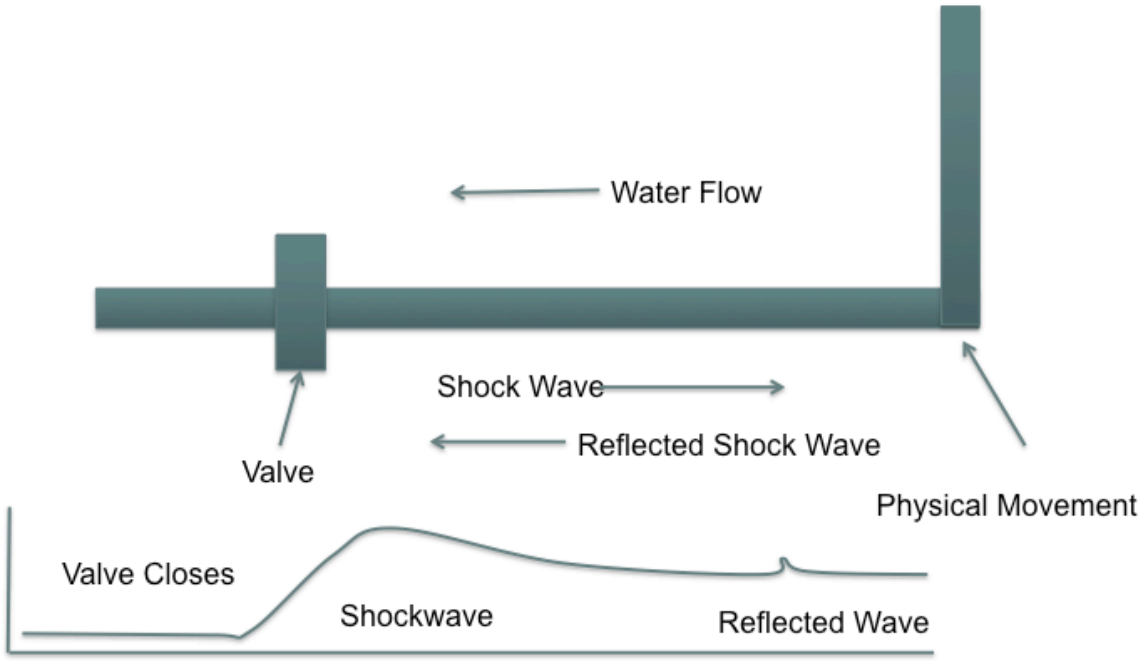


Figure 1 Experimental Set

Figure 1 shows the test setup. A length of pipe containing pressurized water was constructed and a computer-controlled valve was placed at the end. When the valve closes suddenly a pressure transient is generated. This type of pressure transient is sometimes referred to as a water hammer.

The speed of the initial shockwave is well understood. It travels at the speed of sound in the liquid. In this case, the pipe is filled with water so the shockwave will travel at 1497 m/s. What is less understood is the mass movement of water. While the initial shockwave travels at 1497 m/s the mass movement of water propagates at a much slower rate. In a 6-inch pipe at 70 psi, 4 m/s is a reasonable rate. The speed of this mass movement of water can vary greatly based on a number of factors such as water pressure and the diameter of the pipe. Small changes in the experimental setup can significantly affect the rate of the mass movement of water.

The pressure wave travels upstream from the valve and then reflects off various surfaces. The most significant reflection is often the first elbow or joint in the piping structure. The reflected wave travels back towards the valve at a rate similar to the initial wave. If the valve is opened and closed at the right time, a second wave will be generated. A properly timed valve operation will generate a wave that will experience waveform addition with the first wave and the new wave will be larger and more damaging than simply operating the valve a single time. The success of this attack is very dependent on the round trip time of the wave from the valve to the elbow and back again. The valve cycle time must match the period of the wave or the waveform will dissipate.

An attacker that has access to a pressure meter on the pipe would not need to guess the round trip time of the wave. They could simply observe the initial wave and the reflected wave. After observing the two waves a simple formula can be used to calculate the optimal cycle time. If the time X is the time between the valve closing and the shockwave appearing at the sensor and time Y is the time between the initial wave and the reflected wave at the sensor, the optimal cycle time would be 2X+Y. In this attack scenario, the values X and Y are the unknowns and must be extracted from the sensor.

In this scenario, the attack code will be inserted into the pressure meter firmware. The attacker will operate the valve and use the code in the pressure sensor to observe the values X and Y. Since the attacker is going to operate the valve, the operator may be alerted to the attack. The rootkit will also need to spoof the operator during the attack so no alarms are raised.

Part 1 - Making the Attack Code Smaller

Even the best algorithms for inserting the attack code into the firmware aren't going to help if the attack code is too large. It's simply not possible to stick a full copy of OpenSSL and a C library into a microcontroller with only a few kilobytes of memory. This section covers several algorithms that can be used to perform the attack in very few instructions.

Spoofing

The most common method for blinding the operator is the trusty record-and-playback method. If the process is run in batches, the last batch is recorded and played back during the attack. If the process is continuous, the previous day is recorded and played back to the operator. The record and playback method takes an enormous amount of space. The second most common method is to build a model of the process and to use that model to calculate what a normal process would look like and then substituting those calculated values for the real ones. Neither of these methods will work for miniaturized attack code.

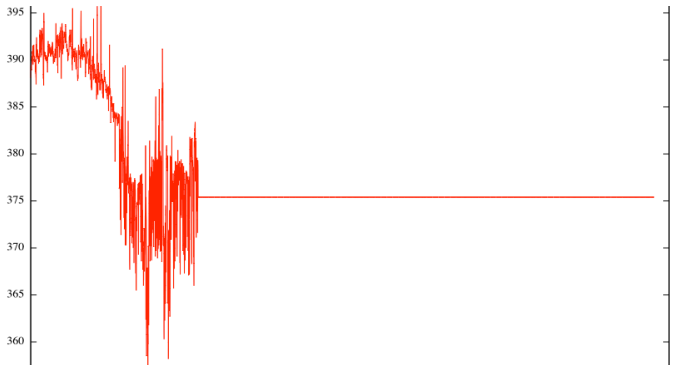


Figure 2 Simply Flat Lining the Sensor Signal

The most trivial approach that would work on microcontroller scales is to simply freeze the data during the attack. The data would be in range, but would no longer have the dynamic behavior of the real data. This approach has very little chance of holding up to forensics investigation after the attack. After all, someone is going to look at the data. The piping structure just mysteriously failed.

Scaling and shifting the data can be used to perform the spoof. Both are small and can give good results in particular cases, but have edge case problems that tend to lead to larger and larger algorithms requiring more and more state data. The attacker is then left with a choice between handling those edge cases and using that room for more important parts of the attack code.

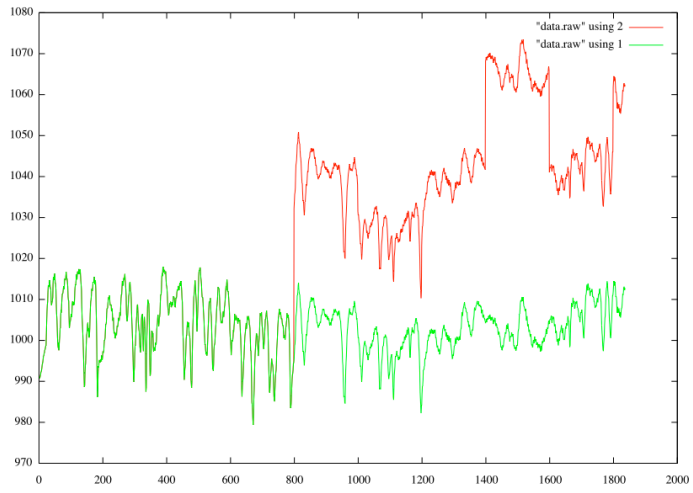


Figure 3 Stair Stepping as the Result of Shifting

Figure 3 shows a typical shifting solution. An average is taken from the signal before the operator starts manipulating the data. The process data is then averaged and periodically a correction factor is calculated between real state of the system and the average. The correction factor cannot be calculated for every sample or the data would simply appear as a flat line. This leads to “stair-step” artifacts in the output data.

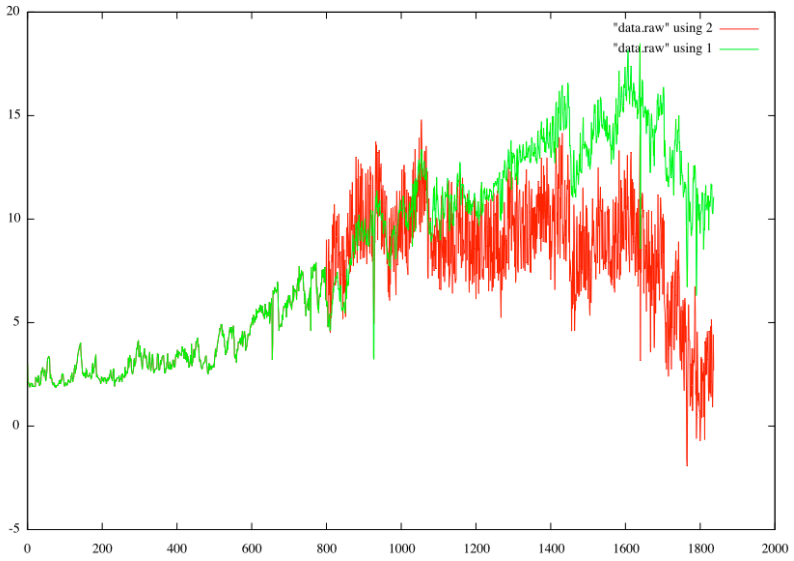


Figure 4 Magnification as the Result of Scaling

Scaling is more of a continuous approach. Again, an average value is taken before the attack. A moving average is then calculated using the process data and is then used to calculate a multiplier between the original average and the observed value. This has the effect of moving all the data into an acceptable range, but has the unwanted side effect of magnifying the size of the noise. Small variations when scaled can become large variations.

Various combinations of shifting and scaling can be combined to produce good a good spoof, but the algorithms tend to break down when given different forms of input. In most attacks, the attacker isn't going to have the luxury of observing the sensor data before the attack and then testing the algorithm to ensure it produces believable data.

The next section describes a new approach that can be deployed without previous knowledge of the sensor noise and that works across a wide range of signals.

Runs Analysis

Runs analysis is a technique used by statisticians to analyze random sequences. The statistician counts the number of increasing/decreasing values in a row. In a perfectly random process, the lengths of the runs should be evenly distributed. The chance of increasing five times in a row should be the same as increasing four times in a row. Since pseudo-random numbers generators are not truly random, there may be a bias towards runs of particular lengths. Two sequences with the same bias might have been generated by the same algorithm.

This technique looks at the sensor noise like a pseudo-random sequence. Since its a bad pseudo-random number generator, we can characterize it by looking at the distribution of runs. Once the sequence is characterized, it can be used to generate sensor noise that is very close to the original noise. It has the added benefit of working across a wide range of sensor noises.

390.3
390.4
390.6
390.3
390.5
390.9
391.1
391.2
390.9
390.9
390.8

Consider the table of observed values above. The first three entries are increasing so the plus_3_count is incremented by one. The total movement over this run is 390.6-290.3=0.3 so the plus_3_movement is incremented by 0.3. The next value (390.3) is decreasing so the minus_1_count is incremented by one and the minus_1_movement is incremented by 0.3. This procedure is repeated during a learning phase noting the length and the movement of each run.

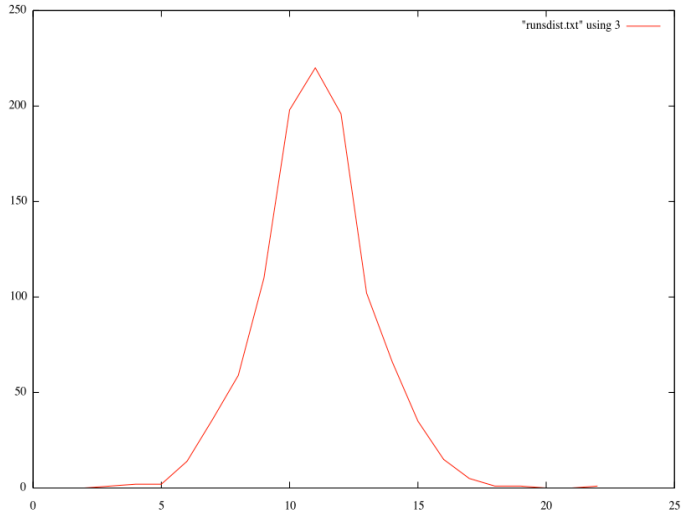


Figure 5 Normal Distribution of the Runs

As expected, the distribution of the runs gives us a normal distribution as shown in Figure 5. After the learning phase, each total movement is divided by its count of the bucket to produce an average movement. Its this table of average movements that contains all the information necessary to generate believable senor noise.

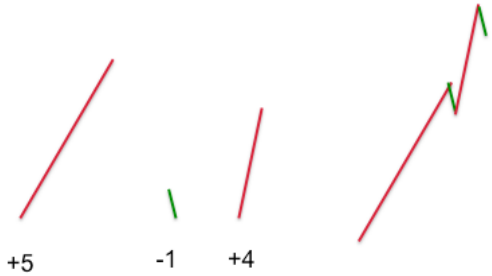
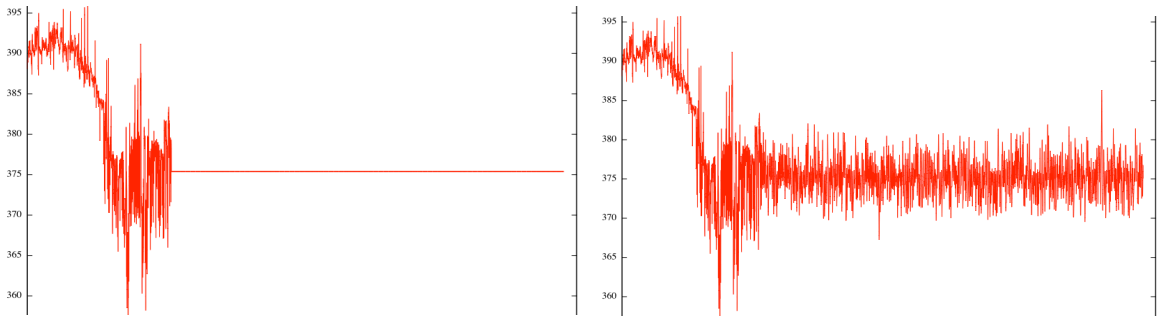


Figure 6 Building Random Noise from the Buckets

Generating sensor noise is now a simple. One of the buckets is chosen at random and a line segment is created from the bucket with the ordinal of the bucket forming the X axis and the average movement number calculated above forming the Y axis. If the desired value is less than the last value shown to the operator, the algorithm chooses from the positive buckets. If the desired value is greater than the last value shown to the operator, the algorithm chooses from the negative buckets. Figure 6 shows four randomly chosen values chained together to form the noise signal from randomly chosen +5,-1,+4,-1. This procedure replicates the original signal “spikiness”, “Gapiness”, and overall line width over a wide range of input signals.



The two figures above show the before-and-after pictures of the runs algorithm. The human important features of the noise are generated in a very believable way. The spoofed signal maintains the same distribution of spikes and gaps. It also maintains the same approximate line width as the original. The noise shown in the first figure is used as the training set. As a result, the generated noise is the average of that training set.

In this experiment, the runs code was hand optimized and required approximately 400 bytes of combined code and data for the entire algorithm. The reason for the approximate value will be covered later in the paper.

Leveling and Artifact Extraction

The Runs algorithm generates believable sensor noise, but it does not solve the problem of what general trend should be shown to the operator. Sensor noise in a flat line may not be the optimal strategy. Other artifacts may need to be shown to the operator. If a pump kicks on when the pressure drops, that pump should be visible in the spoofed data. It is these artifacts that forensics teams are going to use to match up data after the attack is done. The problem facing the attacker is how can those artifacts be extracted from the overall trend of the process. Also, some of those very artifacts are going to be used by the attacker to calculate the round trip time of the shockwave. Figure 7 shows an example of a trend and an artifact.

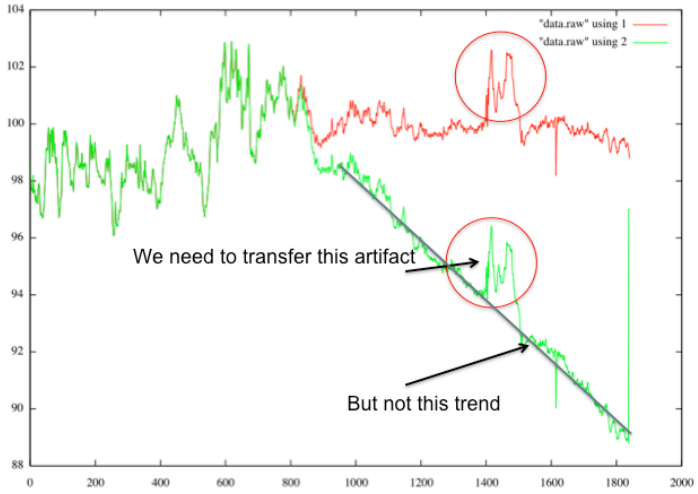


Figure 7 Trend vs. Artifacts in Sensor Data

The most common method used to extract the trend from the artifacts is a simple moving average. The average movement is trend and everything else is either an artifact or noise. Figure 8 shows an example signal that has been filtered using a 2, 50, and 200 point moving average.

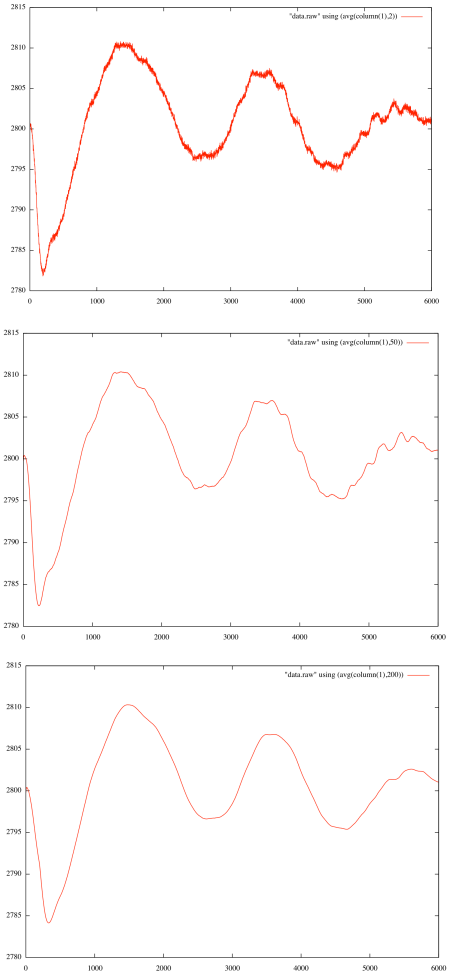


Figure 8 Moving Averages

In this particular example, a 200-point moving average does a reasonable job of smoothing out the signal. Unfortunately, without previous knowledge of the sensor noise, its not possible to know how many points need to be averaged together to smooth out the signal. An algorithm based on a moving average has another big problem. A moving average requires large amounts of memory. The two point moving average only requires 8 bytes of memory to hold the sample, but a 200 point moving average would require 800 bytes just to hold the data. This is clearly unacceptable for insertion into a very small microcontroller.

The next section describes an alternate approach that transforms the signal into a set of best-fit lines that is both small and works across a wide range of signal types.

Triangles

While most computer scientists naturally reach for an FFT and fit sine waves to time-series data, obviously an FFT library is not going to fit into a small microcontroller. A better solution is to fit a set of line segments to the data.

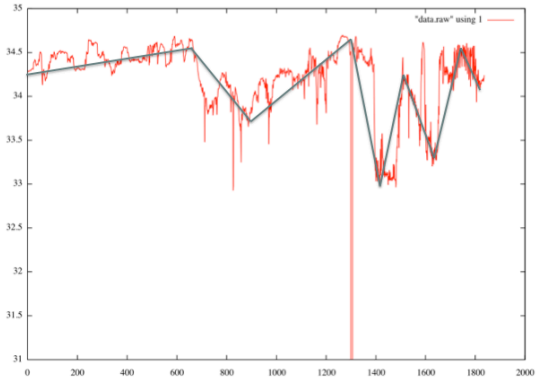


Figure 9 Best-Fit Line Segments

Figure 9 shows temperature data from a buoy off the coast of Japan with a series of line segments fit over the top. The line segments give a good estimation of the dynamic nature of the data. If the algorithm that fits the line segment is dynamic, it will generate more line segment when the data is changing rapidly and fewer line segments when the data is level. Using line segments has an additional benefit. Instead of performing expensive processing operations every sample, those operations can be performed only when a new vertex is declared.

Once a series of line segments is established, removing the trend becomes a trivial process. The sensor noise is simply shifted by distance between the current end of the line segment and the average value calculated earlier. The adjustment is more dynamic, right after a vertex is declared and then less dynamic as the length of the line segment grows effectively transferring the artifacts onto the spoofed signal.

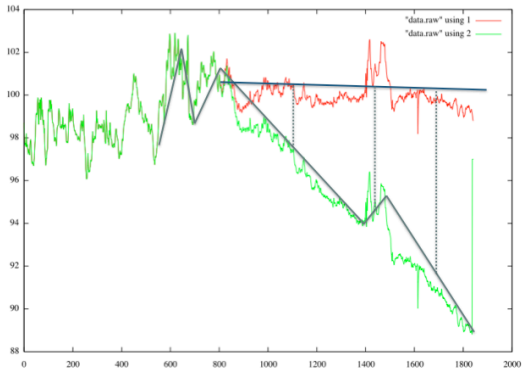


Figure 10 Trend Adjustment via Line Segments

Logic that handles various artifacts is now much simpler. The area of the triangles can be used as an estimate of the area under the curve. Peaks can be trivially found by searching through the vertexes.

The algorithm described below can be used to create the line segments. Conceptually it is based on a triangle starting at the last vertex and growing to the right. Two rays are sent out from the vertex until too many points fall outside the triangle described by the two rays. In that case a vertex is declared, the angle is adjusted and the process starts over.

1) Declare a vertex at the first value.
2) Choose an arbitrary starting window n. Calculate or estimate a smoothing factor s=log(n).
3) Note the minimum and the maximum values in the window.
4) Draw a triangle from the origin through the minimum and maximum values and ending in a vertical line at n.
5) Declare a vertex at the midpoint of the vertical line at n.
6) Start drawing a second triangle from the vertex using the slopes of the previous triangle.
7) Count y,z samples that are above,below the triangle.
8) When y or z > s declare a vertex at the midpoint of the vertical line through the current sample.
9) If y<z increase the slope of the top and decrease the slope of the bottom line otherwise do the opposite.
10) If the number of samples between the current sample and the last vertex < 4n then increase n.
11) 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.
12) Go to step 6 .

There is room for improvement in this algorithm, but it produces good results over a wide range of input types.

Now that artifacts are described by a series of line segments, the attacker needs to find which of those vertexes is actually the peak of the pressure wave. The next section describes a method for sorting through the vertex data to identify the peaks.

Scale Free Artifact Extraction

One of the most expensive parts of detecting an event in the process is scaling the data. If this is pressure data, is it in mm/Hg, cm/H20, or some other scale? Is calibration done on the sensor or in the PLC? Most detection routines use a threshold value that, when exceeded, triggers an action or declares a value used in later actions. In our example, the peak of the first pressure wave and the peak of the second pressure wave are the necessary values to carry out the attack. How do we know when the peaks occur. Changing the data into a set of line segments certainly simplifies the process, but it does not tell us if this particular peak is the pressure wave or just some random local maximum. Many early attempts to solve this problem use a scale to try to differentiate between a local maximum and the true peak. The code necessary to perform this logic easily exceeds all the rest of the code combined.

One solution is to use a scale free matching algorithm to determine if this peak is the correct peak. The size of the peak is not set in absolute units but as a set of ratios with the surrounding peaks. This is can be thought of as a scale-free sequence of events.

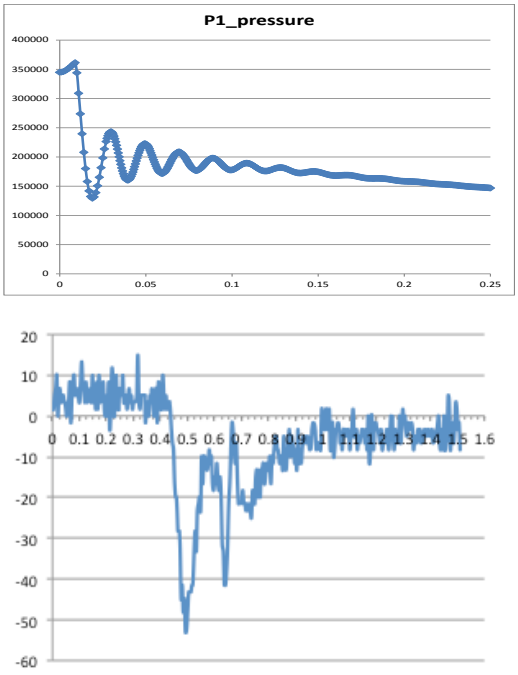


Figure 11 Modeled vs. Observed Pressure Waves

During the experiment, the pressure was modeled using an industry-standard piece of software. Figure 11 shows the pressure waves predicted by a model and the actual measured pressure waves. Manually looking at the data, the valve opens at 0.4 seconds. The first peak is detected at around 0.55 seconds and the first reflection is seen at around 0.65 seconds. So according our formula, the valve should be cycled at (2*(0.55-0.4))+(0.65-0.55)=0.4 seconds to set up a standing wave and damage the piping structure.

Without applying any particular scale to those diagrams, a human can easily see that they match to some degree. If both of those data sets are converted to line segments, they could be compared to see if they were close enough to be considered a match.

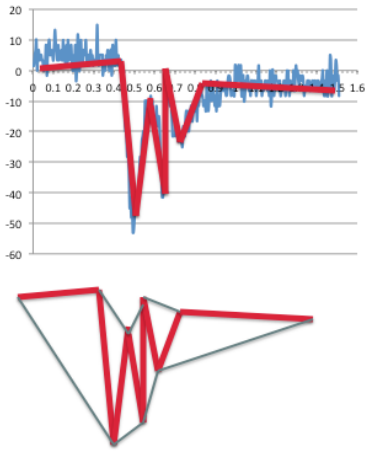


Figure 12 Sensor Data Converted to a Triangle Strip

In this particular experiment, a scale-free algorithm was created by converting the line segments into a triangle strip. The area of each triangle was compared to the area of the adjacent triangle. The ratios between the areas of four adjacent triangles were found to fall within a narrow range even when the parameters of the experiment were manipulated. The scale free description of this event can then be described as a series of four ratios and require only 16 byte of memory to hold.

All of the algorithms necessary to perform the attack have now been described. The operator can be successfully blinded while the peaks of the initial wave and the reflected wave are detected.

Total Size of the Code We Need to Insert
Sensor Noise ~ 400 bytes
Triangles ~ 700 bytes
DNP CRC – 272 bytes
Protocol and Glue Logic ~ 600 bytes
Total Payload – 2174 bytes (about 0.7% of the total flash)

The table above shows the size of the code that will need to be inserted into the target firmware. It requires just over 2 kilobytes of code and data. The next half of this paper will describe a method for efficiently inserting this code into the firmware.

Part 2 - Inserting the Attack Code into the Firmware

Now that a payload as been developed it needs to debugged and inserted into the existing firmware. In some cases, 2 k can be stolen from somewhere. For example, debugging strings could be removed and replaced with code. In many cases however, there is very little extra space in the microcontroller.

The firmware already contains many of the functions the rootkit needs to operate. For example, every piece of code on the planet contains a memcpy() function. On a higher level, if the attacker is planning to use a DNP stack for a control channel, the device probably already contains one. Minimizing the amount of code that needs to be inserted, means reusing any functionality that already exists in the target firmware.

There is a problem with reusing large subsystems in the target firmware. All implementations have side effects. For example, the DNP stack may interact and share code with the IEC870 stack. If the attacker's rootkit calls the DNP stack, he may be inadvertently be adjusting state variables in the IEC870 stack. As the size of the subsystem grows, the probability that there will be side effects increases. Testing for all those side effects can add weeks to the development cycle.

One solution is to write a DNP implementation on a normal PC, debug it, and then efficiently merge it with the target firmware. During the merge, any code in common will be reused but both the rootkit code and the firmware code will run exactly as they were intended. Obviously when constructing the code on the PC, the rootkit creator will want to keep the implementation as close as possible to the target implementation to maximize the code in common.

Comparing Code from Different Architectures

Before a DNP stack written in X86 on a PC can be merged with a DNP stack on an ARM chip, there needs to be a generic way to transform and compare functions from two different architectures.

The comparison and merging process is best described by example. In this example, a simple function is compared on two different architectures: ARM and MSP430. A modified version will be created and then the two versions will be merged together.

```
int CalcSomething(int x){

    int total = 0;

    int i;

    for (i=0;i<x;i++){

            total=total+i;

    }

    return total;

}
```

This function is a simple loop that calculates the sum of a series of numbers. It is a leaf node in that does not call any other functions. This is the simplest test case that can be constructed to illustrate the process. A GCC cross compiler was used to compile it for both of the target architectures.

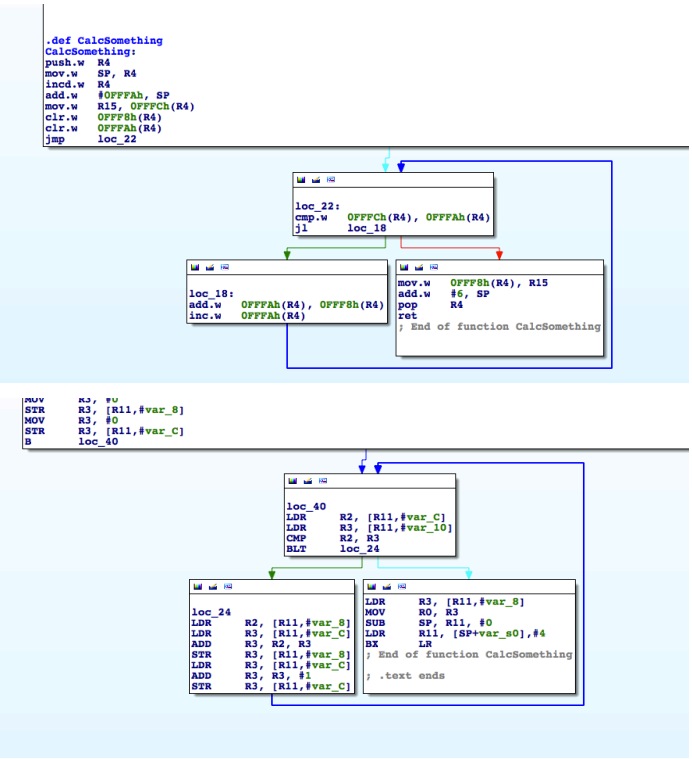


Figure 13 Test Function Compiled for MSP430 and ARM

As shown in Figure 13, the overall structure of the function is the same. It should be since it came from the same C code. The two functions cannot be programmatically compared. First, they need to be converted to a common language.

Micro Operations (MicroOps)

We can start comparing the two assembly languages by breaking the individual instructions into smaller units. Many instructions do several things. For example, the X86 instruction “PUSH EAX” actually performs two smaller operations. It first subtracts 4 from the stack pointer. Secondly, it moves the value in EAX into the memory pointed to by the stack pointer.

PUSH EAX	ESP:=ESP-4
	[ESP]:=EAX

As shown above, the complex instruction is broken down into two MicroOps. Nearly all instructions can be converted to a series of MicroOps that describe everything the instruction does including all of its side effects. For example, the SUB instruction has the side effect of setting several of the flag variables.

The tables below show the assembly instructions from the two functions converted into a series of MicroOps. The first table shows the ARM assembly and the second table shows the MSP430 assembly converted to the equivalent MicroOps. The MicroOps that describe flag manipulation have been removed for the sake of readability.

MOV R3, #0	R3:=0

STR R3, [R11,#var_8]	[R11+8]:=R3
MOV R3, #0	R3:=0
STR R3, [R11,#var_C]	[R11+C]:=R3
B loc_40	PC:=loc_40
LDR R2, [R11,#var_8]	R3:=[R11+8]
LDR R3, [R11,#var_C]	R3:=[R11+C]
ADD R3, R2, R3	R3:=R2+R3
STR R3, [R11,#var_8]	[R11+8:]:=R3
LDR R3, [R11,#var_C]	R3:=R11+C]
ADD R3, R3, #1	R3:=R3+1
STR R3, [R11,#var_C]	[R11+C]:=R3
LDR R2, [R11,#var_C]	R2:=[R11+C]
LDR R3, [R11,#var_10]	R3:=[R11+10]
CMP R2, R3	IF R2< R3 THEN PC:=loc_24
BLT loc_24	

clr.w 0FFF8h(R4)	[R4-8]:=0
clr.w 0FFFAh(R4)	[R4-10]:=0
jmp loc_22	PC:=loc_22
add.w 0FFFAh(R4), 0FFF8h(R4)	[R4+8]:=[R4+10]+[R4+8]
inc.w 0FFFAh(R4)	[R4+10]:=[R4+10]+1
cmp.w 0FFFCh(R4), 0FFFAh(R4)	IF [R4+10]<[R4+8] THEN PC:=loc_18
jl loc_18	

Binary Normal Form

Now that the two functions have been converted to same language, they can be compared. They contain the same logic, but they do not match. The ARM assembly, which was created with a more RISC flavor to it, takes several operations to zero a stack variable while the MSP430 only takes a single instruction.

Since they have already been translated into a common language, it stands to reason that there are a set of transforms that would turn these two sets of MicroOps into an identical set of instructions. This set of rules has been called Binary Normal Form (BNF).

The entire set of rules that comprise BNF are too complex and detailed to go over in this short document, but explaining several of the rules needed to transform the above example is appropriate.

Rule 1 – All Memory Operations Happen via a Single Register

The contents of a memory read operation are stored into a single register. The sources of a memory write operation is a single register. If necessary a phantom temporary register is used.

As an example [0x1000]:=EAX+4 violates Rule 1 because it performs both an ADD and a memory operation in a single MicroOp. It should be transformed into two MicroOps TMP1:=EAX+4;[0x1000]:=TMP1

Rule 2 – All Constants are Loaded via a Single Register

Rule 3 – All Condtional Jumps are Transformed into a Positive Form

There are not “Branch if not equal” operations. They should be transformed into a “Branch if equal” form. The MicroOps should be rearranged so that the true branch always comes before the false branch.

Using these simple sets of rules, the common assembly language can be manipulated so that in most cases, there is only one way to describe a particular set of semantic logic. Obviously, there will always be some cases where equivalent logic will not be transformed into exactly the same form. For example, complex looping structures or inlining a function would obscure an otherwise perfect match.

R3:=0	TMP1:=0
[R11+8]:=R3	[R4+8]:=TMP1
[R11+C]:=R3	[R4+10]:=TMP1
PC:=loc_40	PC:=loc_22
R3:=[R11+8]	TMP1:=[R4+8]
R2:=[R11+C]	TMP2:=[R4+10]
R3:=R2+R3	TMP3:=TMP1+TMP2
[R11+8:]:=R3	[R4+8]:=TMP3
R3:=[R11+C]	TMP1:=[R4+10]
R3:=R3+1	TMP1:=TMP1+1
[R11+C]:=R3	[R4+10]:=TMP1
R2:=[R11+C]	TMP1:=[R4+8]
R3:=[R11+10]	TMP2:=[R4+10]
IF R2< R3 THEN PC:=loc_24	IF TMP1<TMP2 THEN PC:=loc_18

The table above shows the two assemblies that have now been transformed to BNF. The ARM assembly was already mostly RISC in nature so it was changed less than the MSP430 assembly. The two MicroOp lists are now very close to each other, but they are l not exactly the same. The two CPUs have different register sets. Also the compiler has chosen a different order for the arguments on the stack. Visually, it's obvious that [R4+10] is the equivalent of [R11+C] but there are many cases where such an equivalency is not so obvious.

Infinite Register File

If both CPUs had an infinite number of registers, the compiler would have no need for a stack. It would also have no need to spill registers onto the stack or reuse them at all. If the above assembly were transformed with this assumption, the bulk of the matching problems between them would disappear. The rules are simple. A register is never reused. All stack operations are converted to register operations. Any operation that can be removed without changing the semantic meaning of the code is culled.

Loops are the only tricky part. If the rules are strictly applied all simple operations such as R1:=R1+1 would be transformed into R2:=R1+1. The compiler would have no need to reuse R1 in the assembly. However if R1 is the counter in a loop, it needs to be preserved throughout its lifespan. The algorithms for detecting whether a register is a loop register are too long to be covered in this paper but will be covered in a future paper.

S1:=0	S1:=0
S2:=0	S2:=0
PC:=PC+4	PC:=PC+4
S1:=S1+S2	S1:=S1+S2
S2:=S2+1	S2:=S2+1
IF S2< ARG1 THEN PC:=PC-2	IF S2<ARG1 THEN PC:=PC-2

Using an infinite register file eliminates most of the need for a preamble in the function. There is no need to allocate stack variables. There is also no need to save a base pointer for the function. The figure above shows the final transformation of the two assembly languages with an infinite register file. They can be directly compared to show that they are indeed the same logic.

Edit Distances

The two functions from different architectures have been successfully transformed into exactly the same MicroOp list. This great for all the cases where the rootkit code is exactly the same as the target firmware code. Those exact same functions can be detected and then reused eliminating a function from the rootkit. Only reusing functions that are exactly the same is very limiting. A better implementation would be able to merge functions that are *mostly* the same.

Two functions that are only mostly the same can be efficiently merged.

```
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;
}
```

The table above shows a slightly modified function. The total is calculated differently but the general looping structure is the same. How can these two functions be merged in a programmatic way?

```
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
```

The table above shows the two functions converted to BNF. They are mostly the same, but one performs an additional ADD operation. The two pieces of code could be merged together if an IF statement is inserted so the extra ADD is only executed in the evil version. One way to compare any two functions is to consider how many IF statements it would take to make them the same.

```
S1:=0
S2:=0
PC:=loc_40
IF ARG2 THEN
  S1:=S1+S2
ELSE
  S3:=S2+4
  S1:=S1+S3
S2:=S2+1
IF  S2< ARG1 THEN PC:=loc_24
```

The table above shows the merge of the two functions. Since a single IF statement made the logic the same. The two functions can be said to have an edit distance of 1. Obviously, any two functions can be enclosed into one giant IF statement containing the entire bodies of both functions so it is also necessary to calculate the efficiency of the edit distance. Putting a single statement inside an IF block is more efficient than putting the whole function inside the IF statement duplicating the entire function. In the example above, only three instructions are duplicated making it the most efficient edit distance. For the rest of the paper the similarity between to functions will be referred to as the edit distance when the literal meaning would be the edit distance with the fewest duplicated instructions.

Abusing Needleman-Wunsch

In order for this concept to be useful, there needs to be a way to calculate the edit distance between two functions in a programmatic way. Luckily there is a well-known algorithm for calculating the edit distance between two strings. The Needleman-Wunsch algorithm compares two strings and returns the minimum edit distance between them. If the assembly in BNF form is tranformed into a string, Needleman-Wunsch can be used to find that edit distance.

```
Inserting Code is Fun
Inserting Rootkits is Fun
```

```
Inserting _Co___de is Fun
Inserting Rootkits is Fun
```

The figure above shows the results of two test strings using Needleman-Wunsch. Most of the sentence is the same but one word in the middle is different. The algorithm shows that the two strings could be made the same if 2 edits were applied.

Since Needleman-Wunsch doesn't define any particular character set, each MicroOp can be turned into a unique "character". Those characters can be chained together into a string, and those strings can be compared.

S1:=0	MS0	S1:=0	MS0
S2:=0	MS0	S2:=0	MS0
PC:=PC+2	MR+	PC:=PC+3	MR+
S1:=S1+S2	ASS	S3:=S2+4	AS4
S2:=S2+1	AS1	S1:=S1+S3	ASS
IF S2< ARG1 THEN PC:=PC-2	ICLTSAMR-	S2:=S2+1	AS1
		IF S2< ARG1 THEN PC:=PC-3	ICLTSAMR-

The table above show the two functions converted to string tokens. Obviously, these example tokens are meant to be human readable and real tokens would be more complex, but they serve as a good illustration of the process.

MS0MS0MR+ASSAS1ICLTSARM-
MS0MS0MR+AS4ASSAS1ICLTSARM-
MS0MS0MR+___ASSAS1ICLTSARM-
MS0MS0MR+AS4ASSAS1ICLTSARM-

The table above shows the application of Needleman-Wunsch to the two strings. The algorithm correctly calculates an edit distance of 1. It also shows which parts of the two functions will need to be duplicated in order to merge these two functions. A single IF statement can be inserted to make these two functions the same.

```
S1:=0
S2:=0
PC:=PC+6
IF ARG2 THEN
  S1:=S1+S2
ELSE
  S3:=S2+4
  S1:=S1+S3
S2:=S2+1
IF  S2< ARG1 THEN PC:=PC-6
```

Putting It All Together. Merging the Firmware

Now that there is a way to merge any two functions and also calculate how efficient that merge is, it is possible to merge debugged and tested code into the target firmware. First every function in our rootkit is compared to every function in the target firmware. The resulting table is used as the starting place for merge.

Every function could simply be merged with its most efficient match, but chances are if one function matches, a whole routine will match. Starting with the leaf functions, the parents of the leaf functions can be compared to see if they can also be efficiently merged. Repeating this procedure produces a good merge strategy between the rootkit and the target firmware.

After each function in the attack code is merged with a function in the target firmware, the code can be transformed from BNF to the target assembly by reversing the steps. Matching MicroOps to assembly instructions is a complex process but is covered in nearly any compiler textbook.

Firmware Insertion Conclusions

This technique has some distinct advantages. The most notable is that when reusing code in the target firmware, the version of the code in the rootkit does not need to match the version of the code in the firmware exactly. If there are differences between the two versions, those differences will be preserved during the merge process. In this way it is possible to write a rootkit once for a device and then potentially deploy it many times into different versions of the vendor device. This has the potential to greatly reduce the amount of work that a rootkit developer must perform to maintain his tools.

Conclusions

It is possible to construct a full attack on an industrial control system and miniaturize it down into a microcontroller that contains only kilobytes of memory. Using very small algorithms reduces the amount of code that must be inserted. Using new methods to reuse existing code in the target firmware reduces the footprint even more.

Hopefully, this presentation will illustrate the need for change control not just on the PCs and PLCs that run a process, but on the individual devices used to actually control the process. Forensics on PLCs and controllers are now becoming commonplace and it is only a matter of time before the attacker community starting invading individual chips moving one more layer deeper into the process.

The Future – Metasploit for SCADA Firmware

This research opens up an interesting possibility. If each of the techniques above, along with others, were placed into modules, they could form a library of attack payloads. Vendors often reuse code among many devices. For example, it has been shown that many vendors use the same DNP stack. If some work was done to identify the most commonly reused code, they could be placed into a library of targets.

With a library of attack payloads and a library of targets, it would then be possible to construct a toolkit similar to Metasploit where an operator would be able to quickly pick targets and payloads to build attack firmware. It may be possible, in the near future, to perform penetration tests on the actual process as quickly as we now perform penetration tests on Windows® machines.

After a device is identified, the firmware could be retrieved. An automated process could look for common code inside the firmware, and a payload could be configured and inserted into the firmware on the fly. The firmware could then be pushed back into the target device to investigate what happens to the process.

We are not there yet, but I believe we are only a few years away. As always, we live in interesting times.