Securing the network with secure diversified code

July 2020

Alcatel·Lucent Enterprise

Table of Contents

Introduction

Every network manager's nightmare is for an attack on their system that results in losing control of their network or data is compromised. The number of ways that attackers damage a network grows every day, ranging from intrinsic vulnerabilities to compromised back doors.

The Alcatel-Lucent Operating System (AOS) running on the Alcatel-Lucent OmniSwitch® family of switches is hardened with secure diversified code. Secure diversified code uses software diversification and independent verification and validation (IVV) of the AOS to protect networks from intrinsic vulnerabilities, code exploits, embedded malware, and potential back doors that could compromise mission-critical operations. Current and future threats are addressed because the secured, diversified code technology is continuously applied on every new AOS release.

Software diversification is delivered through Address Space Layout Randomization (ASLR). The primary purpose of ASLR is to protect an Alcatel-Lucent OmniSwitch® AOS from a buffer overflow attack. From a technical perspective, a buffer overflow attack is the result of sending a large amount of data to overwhelm the address buffer and then install a carefully crafted packet with a malicious payload (software program). Since the buffer is already full, this malicious packet will be written over other data that an application uses. The packet is designed to contain patterns that force the program to jump to another address location and execute what the attacker desires.

Under IVV, a third party experienced in cybersecurity performs analysis and testing of the AOS switch software to eliminate any potential vulnerabilities, backdoors, malware, or system exploits. The AOS source code is analyzed for these items and is also used to develop source assisted white box and black box vulnerability tests. The tests are subsequently executed on the general availability software images provided by Alcatel-Lucent Enterprise to ensure the integrity of the software.

Software diversification

An attacker needs to know the precise address of the buffer where the packet will be loaded during the buffer overload condition and know the address of the code they need to execute to have the software jump to an address location that enables the malicious program to execute code which changes the operation of the switch. Determining these addresses is a very difficult process requiring much trial and error. However, once these addresses are determined, the attacker develops the malicious packet payload. If they are unable to determine the needed addresses, they cannot develop an attack packet.

Without ASLR, the attacker could do this trial and error process on one switch and once they determine the addresses, apply this attack to all switches running the same software.

ASLR changes the behavior of the virtual memory management system to randomize the location of the different segments on each execution. Code, stack, heap (data) and library segments, will have random locations on each execution. Since the addresses are changed on each bootup, it would be unlikely, or nearly impossible, for an attacker to learn their target addresses by trial and error.

The ALE AOS is extensively tested to reject overflow packets and prevent buffer overflows. ALSR provides an added layer of protection.

AOS ASLR evolution

Moving AOS to ASLR has been a significant development upgrade. Before AOS 8.5.R04, ASLR was enabled for the Linux kernel and some dynamic libraries. The kernel option "randomize_va_space" was enabled. Several dynamically loaded libraries had the "-fPIC" flag set to generate position-independent code. With the "-fPIC" flag set, these libraries are located randomly in virtual address space.

AOS 8.6.R01 completes the ASLR changes. ASLR was enabled on AOS applications using the "-fPIE" compile option. The position independent executable flag allows AOS applications to be in virtual memory according to the Linux kernel randomization. A quick overview of PIE can be found at: http://www.openbsd.org/papers/nycbsdcon08-pie/.

Existing ASLR changes in AOS 8.5.R04

Kernel randomization

Linux randomization can be seen by looking at the randomize flag. The command set below displays the flag.

```
6860E-U28_13_1A_13-> su
Entering maintenance shell. Type "exit" when you are done.
SHASTA #-> sysctl -a |grep  "randomize"
kernel.randomize_va_space = 2
```

The flag definitions are:

```
0 = randomization disabled
1 = randomization enabled (Stack, virtual dynamic shared objects, shared memory)
2 = randomization enabled (1 plus data segments)
```

AOS 8.5.R04 randomization in chassis manager

The randomization of the application segment can be seen by dumping the memory maps for any process between multiple reboots. Task vcmCMM is shown.

Enter the bash shell, if not already in the shell.

```
6860E-U28_13_1A_13-> su
Entering maintenance shell. Type "exit" when you are done.
Dump the process id of the vcmCmm task.
SHASTA #-> ps -ef | grep vcmCmm
2226 root      0:00 /bin/vcmCmm
5666 root      0:00 grep vcmCmm
```

Dump the memory map for the task. Replace 2226 by the task id printed in the above command. The memory ranges of the many libraries are removed from the output listed below for clarity. The segments listed are the ones of importance for this demonstration. These runs were on an OmniSwitch 6860-U28.

```
SHASTA #-> cat /proc/2226/maps
00008000-00275000 r-xp 00000000 00:01 3039      /bin/vcmCmm
0027c000-00299000 rw-p 0026c000 00:01 3039      /bin/vcmCmm
0228b000-022ac000 rw-p 00000000 00:00 0         [heap]
bef63000-befd5000 rw-p 00000000 00:00 0         [stack]
```

Reboot the switch to cause a new randomization. Then repeat the same steps to obtain new addresses. The second bootup address from a sample run are:

```
SHASTA #-> cat /proc/2227/maps
00008000-00275000 r-xp 00000000 00:01 805       /bin/vcmCmm
0027c000-00299000 rw-p 0026c000 00:01 805       /bin/vcmCmm
008fa000-0091b000 rw-p 00000000 00:00 0         [heap]
be8ae000-be920000 rw-p 00000000 00:00 0         [stack]
```

The addresses of the two runs can be compared.

	Run 1	Run 2
/bin/vcmCmm Code segment 1	00008000	00008000
/bin/vcmCmm Code segment 2	0027c000	0027c000
heap	0228b000	008fa000
stack	bef63000	be8ae000

With AOS 8.5.R04 the two code segments are at fixed addresses while the heap and stack are assigned randomly.

Additional ASLR changes in AOS 8.6.R01

Previous ASLR status in AOS 8.6.R01

The same commands used for AOS 8.5.R04 can be used to verify that the Linux kernel, the application stack segment, and heap are randomized in AOS 8.6.R02.

Randomization of the code segment

The final piece of ASLR added in AOS 8.6.R01 was randomizing the code segment address. This can be seen by dumping the code segment address on multiple reboots with AOS 8.6.R02. This is using the same commands as with AOS 8.5.R04.

```
6860E-U28_13_1A_13-> su
Entering maintenance shell. Type “exit” when you are done.
SHASTA #-> ps -ef | grep vcmCmm
 3699 root      0:00 /bin/vcmCmm
SHASTA #-> cat /proc/3699/maps
```

Run 1

```
68c6000-b6b36000 r-xp 00000000 00:01 1537      /bin/vcmCmm
b6b36000-b6b53000 rw-p 00270000 00:01 1537     /bin/vcmCmm
b7738000-b7759000 rw-p 00000000 00:00 0        [heap]
be93e000-be9b0000 rw-p 00000000 00:00 0        [stack]
```

Run 2

```
b693c000-b6bac000 r-xp 00000000 00:01 3460     /bin/vcmCmm
b6bac000-b6bc9000 rw-p 00270000 00:01 3460     /bin/vcmCmm
b7bb2000-b7bd3000 rw-p 00000000 00:00 0        [heap]
be7e4000-be856000 rw-p 00000000 00:00 0        [stack]
```

Comparison:

		Run 1	Run 2
/bin/vcmCmm	Code segment 1	68c6000	b6b36000
/bin/vcmCmm	Code segment 2	b6b3600	b6bac000
heap		b7738000	b7bb2000
stack		be93e000	be7e4000

In the sample runs the code segment is now different between runs.

Deeper dive into the address changes

To better understand how ASLR affects the movement of data, a small program can be executed on the switch. This shows that actual variable locations are changing and not just the segment start. This same detail can be obtained on AOS applications with gdb debugger, but is much harder to visualize. This sample program displays a variable in each of the code, stack and heap segments.

Sample program

```
#include <stdio.h>
#include <stdlib.h>
void* getAddr () {
 return __builtin_return_address(0)-0x5;
};
int main()
{
    void *p = calloc(10000,1);
    void *addr = getAddr();
    printf( "Code: %p\nstack %p\nheap:%p\n",
            addr, __builtin_frame_address(0), p);
    return 0;
}
```

Run without ASLR

The sample program was run on an OmniSwitch 9900. Kernel ASLR is disabled to show the results with no ASLR. The small program is then executed multiple times. All segments have the same address between executions for both the no '-fPIE' and PIE compile programs. An attacker would know that the address would be the same across all switches and reboots.

Command to disable kernel ASLR.

```
MHOST # sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Results from 2 runs of the no PIE and 2 results from the PIE.

	Run 1 no PIE	Run 2 no PIE	Run 3 PIE	Run 4 PIE
Code	0x400586	0x400586	0x555555554886	0x555555554886
Stack	0x7fffffffecd0	0x7fffffffecd0	0x7fffffffecc0	0x7fffffffecc0
Heap	0x601010	0x601010	0x555555755010	0x555555755010

Run with ASLR

The sample program was run on an OmniSwitch 9900. Kernel ASLR is enabled to show the results of kernel ASLR on applications using "-fPIE" and not using PIE. The small program is then executed multiple times. The stack and heap segments are randomized on each execution for all trials. Only the version compiled with "-fPIE" has the code segment randomized.

Command to enable kernel ASLR. Note that AOS 8.6.R02 has kernel randomization on by default.

```
MHOST # sysctl -w kernel.randomize_va_space=2

kernel.randomize_va_space = 2
```

Results from two runs of the no PIE and two results from the PIE.

	Run 1 no PIE	Run 2 no PIE	Run 3 PIE	Run 4 PIE
Code	0x400586	0x400586	0x7f13d8753886	0x7f7124aea886
Stack	0x7ffe987f8930	0x7fff7a97a200	0x7ffeb9aca730	0x7ffcd1b9e8f0
Heap	0x1f30010	0xc43010	0x7f13d8b23010	0x7f71260f2010

Summary

Current and future threats to a network are being addressed by using secured diversified code technology that is applied to every new operating system release from ALE. Alcatel-Lucent Enterprise's secure diversified code uses software diversification and independent verification and validation to protect networks.

Every day there are a growing number of intrinsic vulnerabilities, code exploits and embedded malware as well as potential back doors that can compromise any business. These threats are being stopped with the Alcatel-Lucent Enterprise solution using address space layout randomization that prevents buffer overflow attacks.

Alcatel·Lucent Enterprise