Cisco IOS XR memory forensics analysis

Solal Jacob

Agence Nationale de la Sécurité des Systèmes d'Information

2019

TOC

- I - *IOS XR* internals & forensics analysis
- II - Attack simulation
- III - Detection

I - *IOS XR* internals & forensics analysis

- We would like to be able to analyze a router to know if it was compromised
- For that we want to develop memory forensics tools to detect advanced attack
- *IOS XR* is an exotic system used on core routers

IOS XR

- Used in *Cisco* routers (12000, ASR9000, ...)
- 32 bits version only
- Based on *QNX* 6.4

QNX

- Microkernel released in 1982, now part of *Blackberry*
- Used in embedded system : Routers, Infotainment, Telematics (*Westing House*, *AECL*, *Air traffic Control*, *General Electric*)
- Source was released then closed again

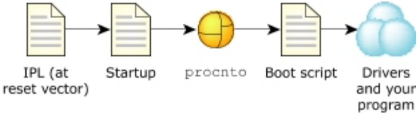
QNX architecture

- Fault tolerant
- Reduced kernel attack surface
- Conforms to posix standard
- Customizable by OEM

QNX Security & Forensics

- Some CVEs
- No hardening before 6.6
- Troopers 2016, QNX : "99 Problems but a Microkernel ain't one !" (Vuln in message passing & IPC)
- Recon 2018, "Dissecting QNX" (Mitigation & PRNG)
- No forensics papers or presentations

QNX startup



- The *IPL*, Inital Program Loader, initializes the hardware, configures the memory controller, loads the system image in *RAM* and jumps to it
- The startup code makes further hardware initilizations, launches the microkernel *procnto* in virtual mode, puts all config info in the system page
- *procnto* runs the boot script and launches other processes (*path manager*, *network stack*, ...)

QNX Firmware

- *IFS* : Image file system, read-only (*procnto*, bootscript, drivers, ...)
- *EFS* : Embedded file system, read-write (program, data, utilities, ...)
- Combined image that can be flashed directly on *NAND*

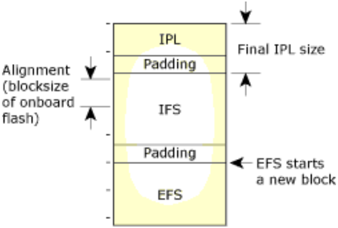


FIGURE – Combined image

- *Blackberry* provides tools to create and read those images

Communication between processes

- *IPC* : Use a message passing system
- Messages are synchronous and directed towards channels and connections rather than threads
- A thread creates a channel to receive messages
- An other thread can make a connection by "attaching" to that channel, then send messages

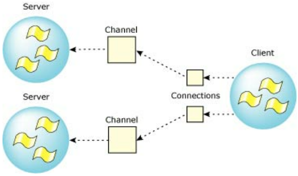
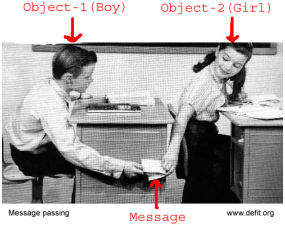


FIGURE – Combined image

The message passing system



- Channels and connections each have an assigned file descriptor
- When a thread creates a channel, it can register a path
- Processes can open these paths via the *path manager*, that returns the file descriptor needed to communicate
- Messages are passed by being copied from the address space of one thread to the address space of an other thread
- There are very few syscalls under QNX 6.4 (~100)
- The libc hides the message passing system like *Linux libc* hides syscalls

Linux libc syscall wrapper

- fd = open("file") ;

```
fd = syscall(open_syscall_number, "file");
```

- write(fd, "abcd", 4) ;

```
ret = syscall(write_syscall_number, fd, "abcd", 4);
```

- close(fd) ;

```
ret = close(close_syscall_number, fd);
```

QNX libc message wrapper

- fd = open("myfile") ;

```
fd = ConnectAttach(PATHMGR_COID, "myfile", 1, 0, 1);

sent_msg.type = IO_CONNECT
sent_msg.data = "myfile"
sent_msg.path_len = strlen("myfile");
MsgSend(fd, sent_msg, sent_msg_size, reply_msg, reply_msg.size);
ConnectDetach(fd);
```

We connect to the service and we ask for a *fd* for this path (*reply->pid* is the pid of the process that handles the hard disks)

```
fd = ConnectAttach(reply->nd, reply->pid, reply->chid, 0, 0);
MsgSend(fd, sent_msg, sent_msg_size, reply_msg, reply_msg_size)
```

QNX libc message wrapper

- write(fd, "abcd", 4) ;

```
sent_msg_buffer.type = IO_WRITE
sent_msg_buffer.nbytes = 4
sent_msg.buffer.data = "abcd"
MsgSend(fd, sent_msg_buffer, sent_msg_buffer_size, ret_msg_buffer, sizeof(
     ret_msg_buffer));
```

- close(fd) ;

```
sent_msg.type = IO_CLOSE
sent_msg.size = sizeof(sent_msg);
ret = MsgSend(fd, sent_msg_buffer, sizeof(sent_msg), 0, 0);
ConnectDetach(fd);
```

Memory acquisition

- Request memory mapping via the memory manager service
- Interfaced via a library call
- All physical memory is directly addressable
- No kernel drivers needed

Memory acquisition tool

- Transfers the memory content via a network socket (to a listening netcat) on *QNX*
- *Cisco* adds its own services and network stack
- They use a modified version of *GCC* to generate specific executables
- A second process manager service is used to launch these executables (They have a *JID* instead of *PID*)
- *Cisco* modified top and other commands to list only applications with a *JID*
- It's difficult to generate a binary that links to the *libsocket* and the *Cisco* network stack
- To create a socket it's possible to use the message system directly

Executing the acquisition tool on *IOS XR*

- The memory acquisition tool can be transfered to *IOS XR* via *ssh*
- It can't be run directly because *Cisco* removed the *chmod* tool
- To made the file executable, we used a trick

Analysis of the memory dump

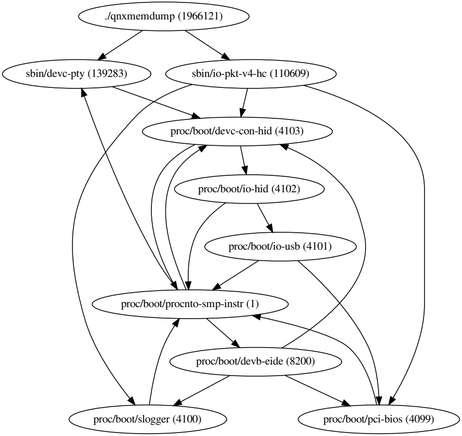
- The *pidin* tool, that lists a lot of system informations, was studied
- It reveals the use of a *syspage_entry* structure, that points to a lot of interesting structures
- To read different structures from the dump we need to know their physical addresses
- The virtual address of the *syspage_entry* struct can be listed via *pidin*
- The *syspage_entry* structure is in the address space of *procnto*
- *procnto cr3* value is needed to convert *syspage_entry* virtual address to physical one

procnto address space

- To find *procnto cr3* value, qemu is used to list the *TLB* entries and find "constant" values
- The system uses identity mapping to translate virtual addresses to physical ones
- The *cr3* value is constant across boots, the value can be found in the *IPL* (that launches *procnto*)
- *procnto* virtual address can be converted to physical thanks to this value
- *syspage_entry* and a lot of user structures can be read
- *Processes*, *memory map*, *channels*, *file descriptor*, ... can then be listed

Connections and channels graph

- *QNX* processes use *IPC*, known as channels, to communicate with other processes
- All the structures containing informations about connections and channels are readable from the memory dumps
- A graph could be created to visualize all the connections
- The graph can be used, for example, to know if a process uses the network stack (Since drivers are processes)



Processes address space

- Each process has its own address space
- To extract each process and its memory map, for further analysis, the physical addresses of its different segments in memory are needed
- For that we need the *cr3* value
- *cr3* is found by following structures linked to *syspage_entry*
- Once we have *cr3* we use it to read the *PTE* and other structures in order to do virtual to physical translation (*PAE* is used in *IOS XR*)
- We can then access all the address space of a process (the segments of the executable mapped in memory and the different allocations made by the process)

Reconstructing the binaries

- Only the data and text segment addresses are listed in *procnto* structures
- Binaries layout differs between *QNX* and *IOS XR* (but QNX binaries are also found in *IOS XR*)
- They are both *ELF*

QNX binaries

- *QNX* binaries are dynamic and have different kinds of segments loaded
- We can't know the address of the *dynamic* segments
- In memory *text* segment is a direct mapping of the offset zero of the binary
- So, it's easy to read the *ELF* header
- The header can be used to rebuild a partial binary containing only the *text* and *data* segments

IOS XR binaries

- The *text* segment is always located at *0x1000* in the binaries (it starts with a *NIAM* header)
- We don't have access to the *ELF* header, it's not mapped in memory
- The binaries are all statically compiled and only have a *text* segment, a *data* segment, an *interpreter* string and an *interpreter* section
- The in-memory *data* segment doesn't have the same size as the one in the binary, so our reconstructed binary will have a different size than the original one
- The interpreter *string* and section contents are always the same
- We can reconstruct an almost complete binary by generating an *ELF* header and then copying the different segments at the right offsets
- The reconstructed binary can then be opened in any disassembler

II - Attack simulation

- No *IOS XR* malware were found to test the detection capabilities of the forensics tools
- We would like to simulate an in-memory attack

Finding an interesting target

- Many *IOS XR* binaries functions contain debug strings with the original name of the function
- We developed an *IDA* script to automatically rename the function to help reverse engineering
- A good target is the *locald* process, a daemon that handles authentication (*ssh*, *telnet*, ...)
- Thanks to our script we easily found the *pw_check* function, an interesting one to modify
- We created a binary with a patched version of this function, so the function will grant access regardless of the password entered
- A user could replace the original binary with this kind of patched binary, but it will be easily detectable

Memory modification of a binary

- To mimic an in-memory attack we created an executable that patches the function directly in memory
- We first need to find the address of the bytes we need to patch
- We use *mmap_device_memory* to give read access to the whole process and find the bytes location
- Then use it again to give write and execute permission to the page that contains the code
- Overwrite the code with our code
- And finally put back the original permissions
- To simulate the attack we executed our binary in a virtual machine
- An attacker could have used the same techniques after exploiting a software vulnerability

III - Detection

- Infect a router in a virtual machine
- Remotely acquire the *RAM* of the router
- Perform a forensics analysis by using our tools and others to identify the attack

Binary diffing

- We would like to compare all the binaries we have extracted from the memory of an the infected router, to the original ones
- The binares are in the firmware images, in the *EFS* partition
- We can extract the partitions from the firmware images with a disk forensics tool
- Then we use Linux *qnx6* file system support to mount the partition read-only
- We then extract all the binaries that are in different directories, each one representing a *package*

Static analysis

- We load each binary in *IDA*
- Apply the script to rename the functions automatically
- Then use a plugin such as *Diaphora* or *Bindiff* to compare our binaries to the ones dumped from memory
- This lets us know if the *text* segment is different between the original binary and the one extracted from memory
- Then it's possible to analyze the differences in each binary in details

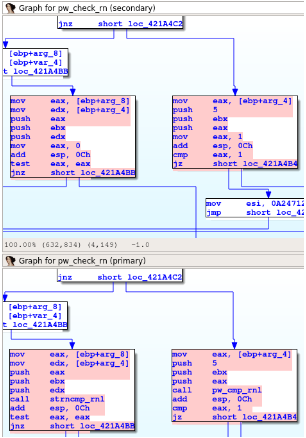


FIGURE – Differences between original and infected *locald*

Dynamic analysis

- We would like to make a dynamic analysis of the reconstructed binary
- The binary can't be run because values in the *data* segment are initialized
- For example the addresses of dynamic libraries

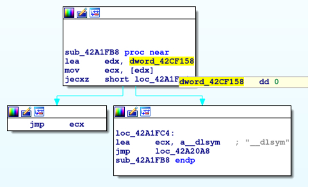


FIGURE – Disassembly of a call to *dlsym*

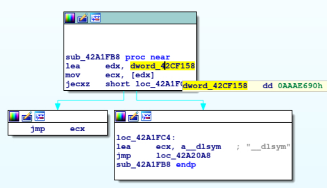


FIGURE – Disassembly of the same function from a reconstructed binary

Automation of the analysis process

- We create a script that follows the traditional forensics model : *preservation*, *collection*, *analysis*, *presentation*
- It periodically launches the memory acquistion tool and stores the dumps
- It then extracts the different processes as *ELF* executables
- Then looks for differences between the router original binary and the one in memory
- Finally it reports the results and warns the investigator if something suspicious is detected
- If something suspicious is found the analyst can go further

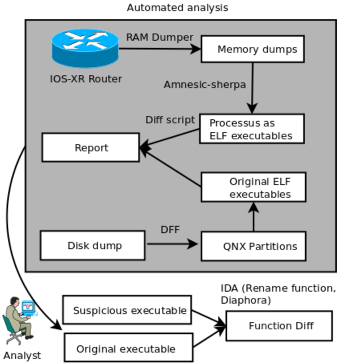


FIGURE – Automated analysis of *Cisco IOS XR*

Conclusion

- We developped a complete forensics & detection framework for *IOS XR* routers
- Our results show that it can detect attacks in an automated way
- We would like to add support for other models of routers and add more functionality
- "Amnesic-Sherpa" the router analysis framework will be available on the ANSSI github
- You can follow me on twitter @ArxSys