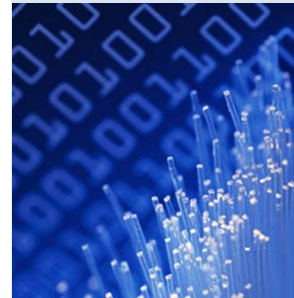




IOS Exploitation Techniques

An IRM Research White Paper by
Gyan Chawdhary



IRM Research

Information technology constantly changes and advances. IRM is dedicated to keeping pace with new technology and continuing to innovate in the field of information security. This ensures that we are well informed of new issues and technologies, expanding our knowledge and providing world class services to our clients.

IOS Exploitation Techniques

It has been more than a year since Michael Lynn first demonstrated a reliable code execution exploit on Cisco IOS at Black Hat 2005. Although his presentation received a lot of media coverage in the security community, very little is known about the attack and the technical details surrounding the IOS *check_heaps()* vulnerability. This paper is a result of research carried out by IRM to analyse and understand the *check_heaps()* attack and its impact on similar embedded devices. Furthermore, it also helps developers understand security-specific issues in embedded environments and developing mitigation strategies for similar vulnerabilities. The paper primarily focuses on the techniques developed for bypassing the *check_heaps()* process, which has traditionally prevented reliable exploitation of memory-based overflows on the IOS platform. Using inbuilt IOS commands, memory dumps and open source tools, IRM was able to recreate the vulnerability in a lab environment. The paper is divided into three sections, which cover the ICMPv6 source-link attack vector, IOS Operating System internal, and finally the analysis of the attack itself.

ICMPv6 Router Solicitation source-link vulnerability

To understand the *check_heaps()* vulnerability we first need to analyse the attack vector used for the exploit. It should be noted that the *check_heaps()* issue exploited by Lynn was highly vulnerability-dependent, as a specific memory layout is required in order to successfully exploit this condition.

The vulnerability used to demonstrate the attack was in the implementation of IPv6 ND (Neighbor Discovery) protocol. The new IPv6 standard introduced the ND protocol, mainly as a replacement for overcoming the design limitations and problems associated with ARP. The ND protocol is primarily responsible for managing all link communications between remote hosts via control message exchanges. These messages provide data necessary for host auto-configuration and utilise ICMPv6 control messages for data exchange.

Without delving into further details of the protocol options, we focus on the ICMPv6 Router Solicitation message which forms the main attack vector for the *check_heaps()* vulnerability. For further information about the ND protocol, the reader should refer to **RFC 2461**.

A Router Solicitation message is generated when a new host is initialised on a network, in order to generate immediate Router Advertisement responses. The packet takes a Type Length Value sub option, where Value is a 128 byte source link IPv6 address. This address is used by the router to determine the physical address of the sending host for generating the correct Router Advertisement responses.

The vulnerability specifically lies in the processing of this sub option, as the length and size fields of the source-link option are not correctly checked by the Neighbor Discovery parser, which allows an internal IOS pool buffer to overflow within the heap memory space. From an attacker's perspective, the vulnerability is quite unique as the size field controls the amount of heap memory to be allocated for our buffer. It was also observed that the overflow was a non-string-based overflow which allowed "null" bytes to be sent in the data stream.

IOS Internals

To fully understand the *check_heaps()* attack we need to familiarise ourselves with the basic IOS subsystem and some of the underlying processes. The *check_heaps()* and watchdog timer are covered below as these processes play a critical role in bypassing the *check_heaps()* process.

The *check_heaps()* Process

As IOS does not deploy full virtual memory support unlike most modern operating systems, there is no concept of process address space segregation for a single running process. This means that all processes share the same memory space and can modify the contents of memory regardless of the privilege associated with them. Furthermore, the above scenario also makes it extremely difficult to debug and detect software bugs and memory leaks in such an environment. As a result, the *check_heaps()* process was introduced to overcome these problems and provide programmers with helpful information for tracing memory leaks and overflows under IOS.

Watch Dog Timer

For process debugging support and CPU resource management, IOS implements a process watchdog timer to detect the presence of unresponsive processes from blocking the CPU resources. When a process is scheduled to run under IOS, the scheduler starts a watch dog timer for the running process. A timeout value of two seconds is used by default before the process timer expired at which point a "SYS-3-CPUHOG" message is generated by the router.

An execution trace of the Watchdog timer expiry event is shown in Figure 1.

```
%SYS-3-CPUHOG: Task ran for 4844 msec (0/0), process = Check heaps, PC = 80475E90.
```

Figure 1: Watch dog timer expiration trace

If the process watchdog timer encounters a second expiry event, the scheduler relinquishes control from the running process. Based on different ISO configurations and the process priority levels, the process is either temporarily suspended or simply terminated.

The ISO *check_heaps()* Attack

The attack involving *check_heaps()* resulted primarily from design issues in the *check_heaps()* error logging functionality and was further exploitable due to the lack of memory protection support between processes. The first ever known exploit to demonstrate memory based code execution under IOS was researched and developed by FX of *Phenoelit*. His technique relied on the fact that an attacker must obtain certain variable values associated with heap management structures in advance to reliably achieve code execution. This was mainly performed to bypass the *check_heaps()* process from detecting memory corruption after an overflow had occurred, which would result in the router being immediately reloaded, thus thwarting all attempts of a successful buffer overflow attack leading to arbitrary code execution. Further information on his techniques is documented in his excellent paper on IOS buffer overflow exploitation in *Phrack 60* article "Burning the bridge".

Like all modern operating systems, IOS implements functionality for logging debugging information in the event of a system crash using inbuilt logging capabilities. The *check_heaps()* process utilises these functions for logging crash-related information after detecting a corrupt memory block. Following is a list of checks performed by *check_heaps()* before the router is loaded:

- Check and log the process which called the *check_heaps()* functions;
- Log all events related to the memory corruption which includes the particular process trace;
- Finally, reboot IOS, based on the switch configuration register, either in "ROMMON" mode or normal configuration.

As noted above, the first check performed by IOS determines the process which invoked *check_heaps()* to flag a memory corruption. This is performed by initializing a Boolean variable which is initially set to zero. For simplicity, let us call this variable *crashing_already*, as described in Lynn's presentation. When the *check_heaps()* process initiates a crash sequence, it checks whether this variable is set to a non-positive value before passing control to the crash logging and debugging functions within the *check_heaps()* functionality. If *crashing_already* has previously been set to a positive value, the *check_heaps()* function simply returns to the caller. This has been implemented as a failsafe mechanism to avoid two concurrent processes from crashing simultaneously, which would otherwise make these error conditions difficult to detect and debug.

As previously mentioned, one of the main motivations for bypassing the *check_heaps()* process was to achieve exploit reliability, while at the same time maintaining access to the system without crashing the device. From an attacker's perspective, the above mechanism can be exploited to achieve this scenario by changing the value of the *crashing_already* variable to a non-zero value. In a default configuration, when the *check_heaps()* process detects a memory corruption, it tries to gracefully shutdown the router. However as the *crashing_already* variable is marked to indicate a crash in progress, every instance of *check_heaps()* would simply return true when an attacker initiated overflow is detected.

Figure 2 shows a flow chart of the *check_heaps()* process.

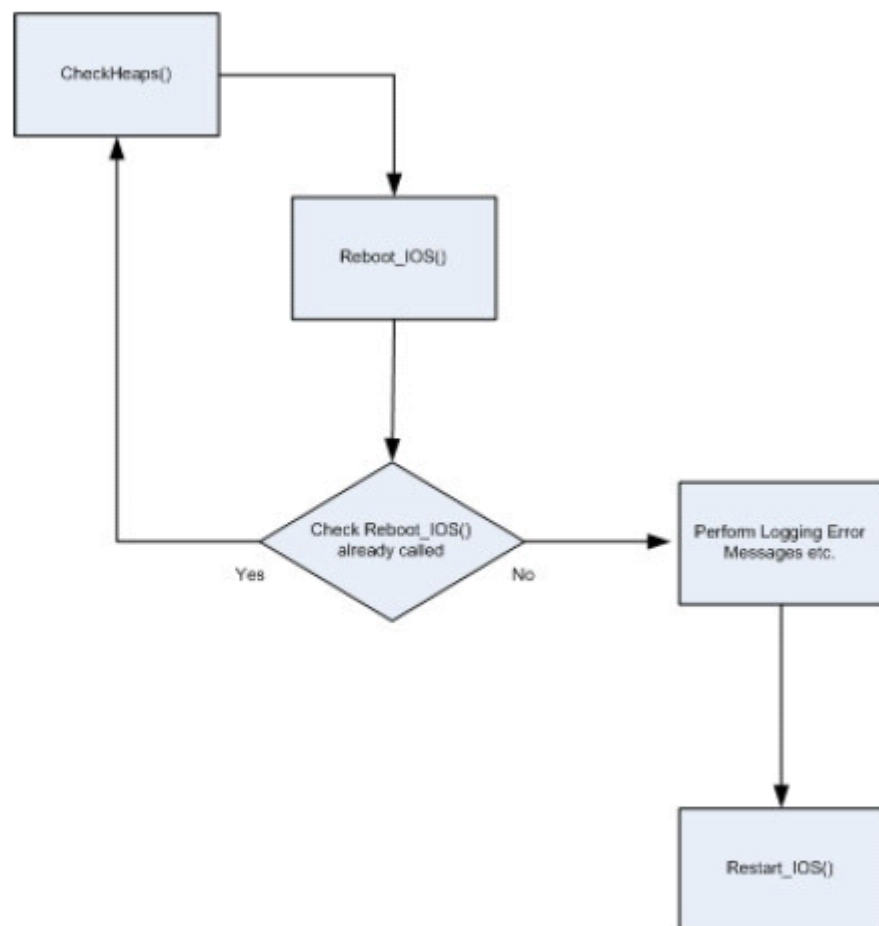


Figure 2: The *check_heaps()* process flow

Now that we know how *check_heaps()* can be bypassed, we can demonstrate the above by overwriting the *crashing_already* flag using the two following techniques:

- Uncontrolled pointer exchange overwrite;
- Kernel timer structure linked lists overwrite.

Uncontrolled pointer exchange technique

As the *check_heaps()* process thoroughly checks the heap "PREVIOUS" pointer to verify memory integrity of the freeing chunk, we can only overwrite up until the "NEXT" pointer in the heap management structure. This provides the opportunity to overwrite an arbitrary value in the user-supplied address when the memory chunk is unlinked. Using this technique an attacker can overwrite the address of *crashing_already* in the "NEXT" pointer, which will result in an arbitrary non-zero value being written to the variable when the chunk is unlinked.

Kernel timer structure linked lists overwrite

A recent Cisco security advisory describes a fix in the operating system timers which allowed code execution on IOS. While debugging the ICMPv6 vulnerability, it was observed that similar error messages were generated related to timer issues. One such error message generated by the crashing device was the "SYS-3-MGDTIMER" error which was further analysed. As shown in Figure 3, the address in the error code (0x82FCBB18) is a pointer to the system timer structure, which remained constant during the tests. Once this was confirmed, several triggers were generated by overwriting specific elements of the timer data structure. Using this technique, the crash dumps were collected and analysed, all returning similar error messages to timer structure corruption. Figure 3 shows some of the timer errors captured during the test.

```

#*Mar 1 00:02:49.515: %SYS-3-MGDTIMER: Uninitialized timer, timer stop, timer = 82FCBB18.
#-Process= "IPv6 ND", ipl= 0, pid= 133
#-Traceback= 80477D34 80478E10 817AE1CC 8048F680 80492BC8
#*Mar 1 00:02:54.499: %SCHED-3-UNEXPECTEDTIMER: Unknown timer expiration, timer = 82FCBB18,
type 16705.
#-Process= "IPv6 ND", ipl= 0, pid= 133
#-Traceback= 817AE1C4 8048F680 80492BC8

#*Mar 1 00:00:39.911: %SYS-3-MGDTIMER: Timer not a leaf, set_exptime, timer = 82FCBB18.
#-Process= "IPv6 Input", ipl= 0, pid= 84
#-Traceback= 80477D78 80478500 80478610 817AA414 817AC184 817ACEB4 817B48C4 817B1CCC 817B1FB0
817B18F4 817B13BC 817B8F24 80488

#*Mar 1 00:00:44.895: %SCHED-3-STUCKMTMR: Sleep with expired managed timer 0, time 0xAF64
(00:00:00 ago).
#-Process= "IPv6 ND", ipl= 6, pid= 133
#-Traceback= 8047FEFC 804802BC 817AE060 8048F680 80492BC8

#*Mar 1 00:04:42.023: %SCHED-3-UNEXPECTEDTIMER: Unknown timer expiration, timer = 82FCBB18,
type 16705.
#-Process= "IPv6 ND", ipl= 0, pid= 133
#-Traceback= 817AE1C4 8048F680 80492BC8
#*Mar 1 00:04:47.039: %SYS-3-MGDTIMER: Uninitialized timer, timer stop, timer = 82FCBB18.
#-Process= "IPv6 ND", ipl= 0, pid= 133
#-Traceback= 80477D34 80478E10 817AE1CC 8048F680 80492BC8

```

Figure 3: Captured timer errors

The data surrounding the timer pointer was further analysed using the *show memory* and *show context* commands. The memory dump revealed (Figure 4) these addresses pointed to similar structures in memory, suggestive of an IOS timer linked list data structure.

```
#82FCBB10:          00000000 82FCBB18          ....|;.
#82FCBB20: 82D89218 82FCBAE8 00000000 004B4760 .X...|:h....KG'
#82FCBB30: 00014240 00000000 00000000 004B33D8 ..B@.....K3X
#82FCBB40: 00000000 004B33D8 00000004 00000000 .....K3X.....
#82FCBB50: 00000000 00000000 00000000 00000000 .....
#82FCBB60: 00000000 00000000 00000000 00000000 .....
#82FCBB70: 00000000 00000000 00000000 00000000 .....
#82FCBB80: 00000000 00000000 00000000 00000000 .....
```

Figure 4: Memory dump indicating timer linked list data structure

IOS system timers are similar to the UNIX timer implementation and managed in a linked list for some particular processes. By overwriting the timer linked list with attacker supplied data, it might be possible to achieve code execution, which was tested by manipulating these addresses. It was observed that the fourth element of the timer structure was processed by IOS and modified by some timer functions. To test this, the structure was replaced by the address of the *crashing_already* address using the ICMPv6 vulnerability as an overflow vector. It was observed that this structure was later processed by IOS resulting in the *crashing_already* address to change into a positive number. When the *check_heaps()* process detected memory corruption due to our overflow, the router initiated the crashing process. However as the *crashing_already* variable had been overwritten, after dumping the contents of the memory, it resumed normal operation without rebooting the router. A screenshot of the router console output is attached in Appendix A.

It should be noted that this technique was only used to demonstrate the bypassing of the *check_heaps()* process. Furthermore, it is possible to exploit these timer structures by overwriting context pointers and callback information to achieve complete code execution; however details regarding this are beyond the scope of this paper.

Conclusions

The *check_heaps()* vulnerability was mainly due to design issues and further exploitable due to the lack of memory protection support between processes. Many embedded system vendors still rely on choosing performance and speed over security. As more and more “intelligence” is built into consumer and commercial devices using embedded operating systems and software, the more significant these potential vulnerabilities may become. Therefore, embedded systems vendors need to be aware of the potential attacks against their systems and the fact that many hackers are getting bored with researching traditional operating systems and are turning to embedded devices for a new challenge.

About the Author

Gyan Chawdhary is a Senior Security Consultant at Information Risk Management Plc (IRM) where he heads up the Embedded Systems Centre of Excellence, and is based in IRM’s European Technical Centre, Cheltenham, England. Gyan has six years of security consultancy experience in Europe, the Middle East and Asia including vulnerability research, reverse engineering and source code audit. He has also publicly released exploit code for PHP and Sendmail vulnerabilities.

About IRM

Information Risk Management Plc (IRM) is a vendor independent information risk consultancy founded in 1998. IRM has become a leader in client side risk assessment, technical level auditing and in the research and development of security vulnerabilities and tools. IRM is headquartered in London with Technical Centres in Europe and Asia as well as Regional Offices in the Far East and North America. Please visit our website at www.irmplc.com for further information.