WHITE PAPER FOR PUBLIC DISTRIBUTION

InformationRiskManagement

IOS Exploitation Techniques

An IRM Research White Paper by

Gyan Chawdhary

Information Risk Management Plc 8th Floor Kings Building Smith Square London SW1P 3JJ
Tel: +44 (0)20 7808 6420 Fax: +44 (0)20 7808 6421 info@irmplc.com http://www.irmplc.com

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 in three sections, which cover the ICMPv6 source-link attack vector, IOS Operating System internals, 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-dependant, 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 messages 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 process 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 expires 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 IOS configurations and the process priority levels, the process is either temporarily suspended or simply terminated.

The IOS 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 reloaded

- Check and log the process which called the *check_heaps()* function
- 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 initialising 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 mentioned previously, 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.

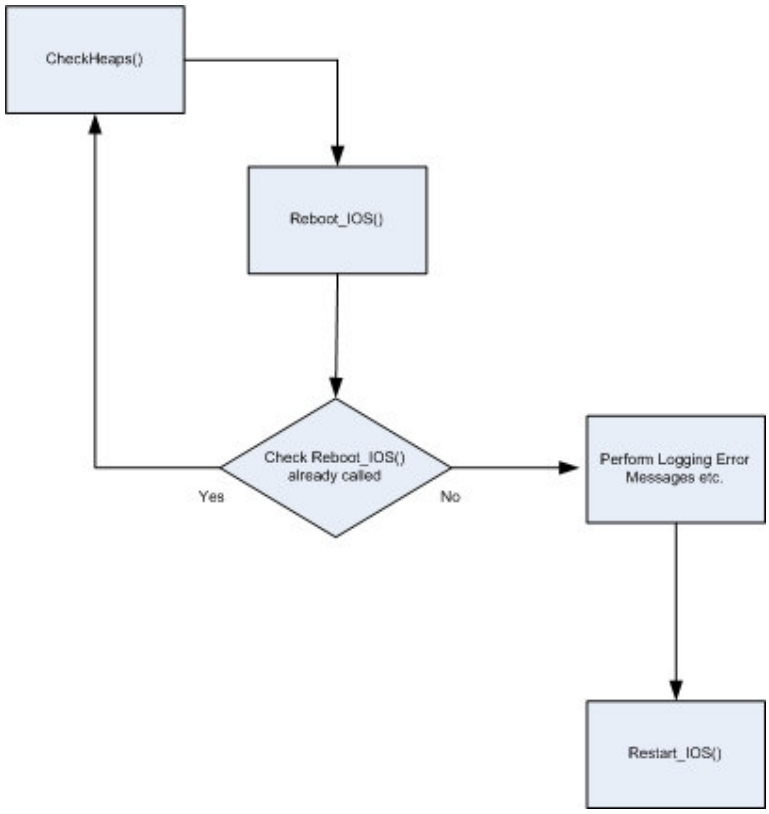

Figure 2: The check_heaps() process flow

Now that we know how *check_heaps()* can be bypassed, we can demonstrate the above by overwriting the *crashing_already* flag using the following two 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 (0x*82FCBB18)* 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 related 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 (Refer to 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 my 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.

Appendix A

```
#attack#
#attack#
#*Mar  1 00:01:53.043: %SYS-3-MGDTIMER: Parent is a leaf, set_exptime_internal, timer
= 82FCBB18.
#-Process= "IPv6 Input", ipl= 0, pid= 84
#-Traceback= 80477DBC 80478090 80478508 80478610 817AA414 817AC184 817ACEB4 817B48C4
817B1CCC 817B1FB0 817B18F4 817B13BC 817B8F24 8048F680 80492BC8
#========= Dump bp = 82FC895C ======================
#
#82FC885C: 82FC8868 82B24504 82E90FF8 82FC8858 82FC8878 82B244D0 82E90FF8 82FC8868
#82FC887C: 82FC8888 82B2449C 82E90FF8 82FC8878 82FC8898 82B24468 82E90FF8 82FC8888
#82FC889C: 82FC88A8 82B24434 82E90FF8 82FC8898 82FC88B8 82B24400 82E90FF8 82FC88A8
#82FC88BC: 82FC88C8 82B243CC 82E90FF8 82FC88B8 82FC88D8 82B24398 82E90FF8 82FC88C8
#82FC88DC: 82FC88E8 82B24364 82E90FF8 82FC88D8 82FC88F8 82B24330 82E90FF8 82FC88E8
#82FC88FC: 82FC8908 82B242FC 82E90FF8 82FC88F8 82FC8918 82B242C8 82E90FF8 82FC8908
#82FC891C: 82FC8928 82B24294 82E90FF8 82FC8918 82FC8938 82B24260 82E90FF8 82FC8928
#82FC893C: 82FC7EE0 82B24228 82E90FF8        0        0        0        0 FD0110DF
#82FC895C: AB1234CD       54 8334B7CC 826282B4 817AA7B8 82FCBBB8 82FC8368 8000191A
#82FC897C:        2        0        0        0        0        0        0       40
#82FC899C:       60   700080       5F        1        1       5F       5C   210000
#82FC89BC: 82CE7B2C 49507636 204E4420 7461626C 65000000 82FC8B58 82FCBB58 82FC8B68
#82FC89DC: 82FC8BE8 82FC8C68 82FC8CE8 82FC8D68 82FC8DE8 82FC8E68 82FC8EE8 82FC8F68
#82FC89FC: 82FC8FE8 82FC9068 82FC90E8 82FC9168 82FC91E8 82FC9268 82FC92E8 82FC9368
#82FC8A1C: 82FC93E8 82FC9468 82FC94E8 82FC9568 82FC95E8 82FC9668 82FC96E8 82FC9768
#82FC8A3C: 82FC97E8 82FC9868 82FC98E8 82FC9968 82FC99E8 82FC9A68 82FC9AE8 82FC9B68
#========= Dump bp->next = 82FCBBB8 ======================
#
#82FCBAB8:        0        0        0        0        0        0        0        0
#82FCBAD8: 15A3C78B        1 817AA860 82FC8984        0 830FF914 20000001    10001
#82FCBAF8:  2112FFF FE14E6F2  1414141 44444444 44444444 44444444 44444444 44444444
#82FCBB18:        0 82FCBB18 82FCBAE8 828DD61B        0  294E794    14240        0
#82FCBB38:        0 20202020        0    1B998        3 BBBBBBBB CCCCCCCC 80818283
#82FCBB58:  80CBB76 808A8B8C 51515151 52525252       20 828DD684 828DD684 828DD684
#82FCBB78: 828DD684 828DD684 828DD684 828DD684 828DD684 828DD684 828DD684 828DD684
#82FCBB98: 828DD684 828DD684 828DD684 828DD684 828DD684 828DD684 828DD684 FD0111DF
#82FCBBB8: AB1234CD FFFFFFFF        0 81DA6944 80487020 82FCBC10 82FC8970 80000018
#82FCBBD8:        1        0 832BDE14      BB8      A2C 82466764        0        0
#82FCBBF8:        0        0        0        0        0 FD0110DF AB1234CD FFFFFFFE
#82FCBC18:        0 81DA7020 80484554 82FCC40C 82FCBBCC 800003EA        1 8046650C
#82FCBC38: 82FCC434 83460658        0        0 83032FE8       64       77    C000C
#82FCBC58:        0     1115     1115       1E        C    10000 82CE7B2C 52656720
#82FCBC78: 46756E63 74696F6E 20310000 82FCBE68 82FCC3FC 82FCBE68 82FCBE74 82FCBE80
#82FCBC98: 82FCBE8C 82FCBE98 82FCBEA4 82FCBEB0 82FCBEBC 82FCBEC8 82FCBED4 82FCBEE0
#========== Dump bp->previous = 82FC8368 ======================
#
#82FC8268: 82B235FC 82E90FF8 82FC8260 82FC8280 82B235C8 82E90FF8 82FC8270 82FC8290
#82FC8288: 82B23594 82E90FF8 82FC8280 82FC82A0 82B23560 82E90FF8 82FC8290 82FC82B0
#82FC82A8: 82B2352C 82E90FF8 82FC82A0 82FC82C0 82B234F8 82E90FF8 82FC82B0 82FC82D0
#82FC82C8: 82B234C4 82E90FF8 82FC82C0 82FC82E0 82B23490 82E90FF8 82FC82D0 82FC82F0
#82FC82E8: 82B23414 82E90FF8 82FC82E0 82FC8300 82B233E0 82E90FF8 82FC82F0 82FC8310
#82FC8308: 82B233AC 82E90FF8 82FC8300 82FC8320 82B23378 82E90FF8 82FC8310 82FC8330
#82FC8328: 82B23344 82E90FF8 82FC8320 82FC78D8 82B23310 82E90FF8        0        0
#82FC8348:        0        0 FD0110DF AB1234CD FFFFFFFE        0 82FAF4F0 80476940
#82FC8368: 82FC895C 82FC7D60 800002F0        1 804664F4        0 82FC22FC 82FC776C
#82FC8388: 82FAF328 82FAF4B4       32       46   100010        0        0        0
#82FC83A8:        0       10   830000 82CE7B2C 4C697374 20456C65 6D656E74 73000000
```

```
#82FC83C8: 82FC84E8 82FC8948 82FC84E8 82FC84F8 82FC8508 82FC8518 82FC8528 82FC8538
#82FC83E8: 82FC8548 82FC8558 82FC8568 82FC8578 82FC8588 82FC8598 82FC85A8 82FC85B8
#82FC8408: 82FC85C8 82FC85D8 82FC85E8 82FC85F8 82FC8608 82FC8618 82FC8628 82FC8638
#82FC8428: 82FC8648 82FC8658 82FC8668 82FC8678 82FC8688 82FC8698 82FC86A8 82FC86B8
#82FC8448: 82FC86C8 82FC86D8 82FC86E8 82FC86F8 82FC8708 82FC8718 82FC8728 82FC8738
#============================================
#
#*Mar  1 00:02:03.043: %SYS-3-MGDTIMER: Uninitialized timer, timer stop, timer =
828DD64B.
#-Process= "IPv6 ND", ipl= 0, pid= 133
#-Traceback= 80477D34 80478E10 817ADB94 817AE0E4 8048F680 80492BC8
#*Mar  1 00:02:04.039: validblock_diagnose, code = 1
#*Mar  1 00:02:04.039:
#current memory block, bp = 0x82FC895C,
#memory pool type is Processor
#*Mar  1 00:02:04.039: data check, ptr = 0x82FC8984
#*Mar  1 00:02:04.039:
#next memory block, bp = 0x82FCBBB8,
#memory pool type is Processor
#*Mar  1 00:02:04.039: data check, ptr = 0x82FCBBE0
#*Mar  1 00:02:04.039:
#previous memory block, bp = 0x82FC8354,
#memory pool type is Processor
#*Mar  1 00:02:04.039: data check, ptr = 0x82FC837C
#*Mar  1 00:02:08.867: %SYS-3-OVERRUN: Block overrun at 82FC895C (red zone FD0111DF)
#-Traceback= 8047383C 80470E44 80466510 80475E7C 8048F680 80492BC8
#*Mar  1 00:02:08.867: %SYS-6-MTRACE: mallocfree: addr, pc
# 82FC8984,8046650C 8351C6C0,8046650C 831ED6C8,8046650C 834A0F9C,8046650C
# 832BFFBC,8046650C 831EB8D4,8046650C 8349E860,8046650C 8349C124,8046650C
#*Mar  1 00:02:08.867: %SYS-6-MTRACE: mallocfree: addr, pc
# 8348A6D4,8046650C 834DBD64,8046650C 8348B54C,8046650C 8347AA78,8046650C
# 8347833C,8046650C 8346D04C,8046650C 8346CA44,8046650C 8345EADC,8046650C
#*Mar  1 00:02:08.867: %SYS-6-BLKINFO: Corrupted redzone blk 82FC895C, words 6426,
alloc 817AA7B8, InUse, dealloc 0, rfcnt 2
#-Traceback= 8046A574 8047384C 80470E44 80466510 80475E7C 8048F680 80492BC8
#*Mar  1 00:02:08.871: %SYS-6-MEMDUMP: 0x82FC895C: 0xAB1234CD 0x54 0x8334B7CC
0x826282B4
#*Mar  1 00:02:08.871: %SYS-6-MEMDUMP: 0x82FC896C: 0x817AA7B8 0x82FCBBB8 0x82FC8368
0x8000191A
#*Mar  1 00:02:08.871: %SYS-6-MEMDUMP: 0x82FC897C: 0x2 0x0 0x0 0x0
#*Mar  1 00:02:08.871: %SYS-3-CPUHOG: Task ran for 4844 msec (0/0), process = Check
heaps, PC = 80475E90.
#-Traceback= 80475E94 8048F680 80492BC8
#attack#
#attack#
#attack#show memory 0x828dd684
#%% Low on memory; try again later
#828dd684: 0x00300000 0x00000000 0x00000000
#%% Low on memory; try again later
#attack#
```