14:06 How likely are random bytes to be a NOP sled on ARM?

by Niek Timmers and Albert Spruyt

Howdy folks!

Any of you ever wondered what the probability is for executing random bytes in order to do something useful? We certainly do. The team responsible for analyzing the Nintendo 3DS might have wondered about an answer when they identified the 1st stage boot loader of the security processor is only encrypted and not authenticated.[14] This allowed them to execute random bytes in the security processor by changing the original unauthenticated, but encrypted, image. Using a trial and error approach, they were able to get lucky when the image decrypts into code that jumps to a memory location preloaded with arbitrary code. Game over for the Nintendo 3DS security processor.

We generalize the potential attack primitive of executing random bytes by focusing on one question: What is the probability of executing random bytes in a NOP-like fashion? NOP-like instructions are those that do not impair the program's continuation, such as by crashing or looping.

Writing NOPs into a code region is a powerful method which potentially allows full control over the system's execution. For example, the NOPs can be used to remove a length check, leading to an exploitable buffer overflow. One can imagine various practical scenarios to leverage this attack primitive, both during boot and runtime of the system.

A practical scenario during boot is related to a common feature implemented by secure embedded devices: Secure Boot. This feature provides integrity and confidentiality of code stored in external flash. Such implementations are compromised using software attacks[15] and hardware attacks.[16] Depending on the implementation, it may be possible to bypass the authentication but not the decryption. In such a situation, similar to the Nintendo 3DS, changing the original encrypted image will lead to the execution of randomized bytes as the decryption key is likely unknown.

During runtime, secure embedded devices often provide hardware cryptographic accelerators that implement Direct Memory Access (DMA). This functionality allows on-the-fly decryption of memory from location A to location B. It is of utmost importance to implement proper restrictions to prevent unprivileged entities from overwriting security sensitive memory locations, such as code regions. When such restrictions are implemented incorrectly, it potentially leads to copying random bytes into code regions.

The block size of the cipher impacts the size directly: 8 bytes for T/DES and 16 bytes for AES. Additionally the cipher mode has an impact. When the image is decrypted using ECB, an entire block will be pseudo randomized without propagating to other blocks. When the image is decrypted using CBC, an entire block will be pseudo randomized. Additionally, any changes in a cipher block will propagate directly into the plain text of the subsequent block. In other words, flipping a bit in the cipher text will flip the bit at the same position in the plain text of the subsequent block. This allows small modifications of the original plain text code which potential leads to arbitrary code execution. Further details for such attacks are for another time.

The pseudo random bytes executed in these scenarios must be executed in a NOP-like fashion. This means they need too be decoded into: valid instructions and have no side-effect on the program's continuation. The amount of different instruction matching these requirements are target dependent. Whenever these requirements are not met, the device will likely crash.

We approximated the probability for executing random bytes in a NOP-like fashion for Thumb and ARM and under different conditions: QEMU, native user and native bare-metal. For each execution, the probability is approximated for executing 4, 8 and 16 random bytes. Other architectures or execution states are not considered here.

[14] *Arm9LoaderHax – Deeper Inside* by Jason Dellaluce

[15] *Amlogic S905 SoC: bypassing the (not so) Secure Boot to dump the BootROM* by Frédéric Basse

[16] *Bypassing Secure Boot using Fault Injection* by Niek Timmers and Albert Spruyt at Black Hat Europe 2016

Executing in QEMU

The probability of executing random bytes in a NOP-like fashion is determined using two pieces of software: a Python wrapper and an Thumb/ARM binary containing NOPs to be overwritten.

```
void main (void) {
  ...
  printf("FREE ");
  asm volatile (
    "mov r1, r1"; // Place holder bytes
    "mov r1, r1"; // ""
    "mov r1, r1"; // ""
    "mov r1, r1"; // ""
  );
  printf("BEER!");
  ...
}
```

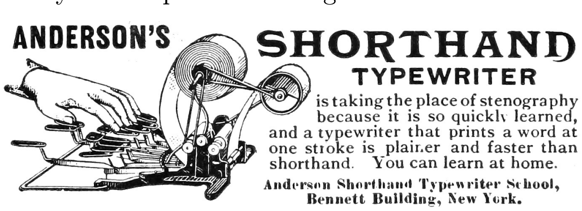
This is cross compiled for Thumb and ARM, then executed in QEMU.

```
arm-linux-gnueabihf-gcc -o test-arm \
        test-arm.c -static -marm (-mthumb)
qemu-arm test-arm
```

Whenever the test program prints "FREE BEER!" the instructions executed between the two `printf` calls do not impact the program's execution negatively; that is, the instructions are NOP-like. The Python wrapper updates the place holder bytes with random bytes, executes the binary, and logs the printed result.

The random bytes originate from `/dev/urandom`. Executing the updated binary results in: intended (NOP-like) executions, unintended executions (e.g. only "FREE" is printed) and crashes. The results of executing the binary ten thousand times, grouped by type, are shown in Table 1. A small percentage of the results are unclassified.

The results show that executing random bytes in a NOP-like fashion has potential for emulated Thumb/ARM code. The amount of random bytes impact the probability directly. The density of bad instructions, where the program crashes, is higher for Thumb than for ARM. Let's see if the same probability holds up for executing native code.



Cortex A9 as a Native User

The binary used to approximate the probability on a native platform in user mode is similar as listed in Section 2. Differently, this code is executed natively on an ARM Cortex-A9 development board. The code is developed, compiled and executing within the Ubuntu 14.04 LTS operating system. A disassembled representation of the ARM binary is shown below:

```
10804:   e92d4800   push   {fp, lr}
10808:   e28db004   add    fp, sp, #4
1080c:   ebfffff0   bl     107d4 <p1>
// These bytes are updated by the
// python wrapper before each execution.
10810:   e1a01001   mov    r1, r1
10814:   e1a01001   mov    r1, r1
10818:   e1a01001   mov    r1, r1
1081c:   e1a01001   mov    r1, r1
10820:   ebfffff1   bl     107ec <p2>
10824:   e8bd8800   pop    {fp, pc}
```

The results of performing one thousand experiments are listed in Table 2.

The results show that executing random bytes in a NOP-like fashion is very similar between emulated code and native user mode code. Let's see if the same probability holds up for executing baremetal code.

Cortex A9 as Native Bare Metal

The binary used to approximate the probability on native platform in bare metal mode is implemented in U-Boot. The code is very similar to that which we used on Qemu and in userland. U-Boot is only executed during boot and therefore the platform is executed before each experiment. The target's serial interface is used for communication. A new command is added to U-Boot which is able to receive random bytes via the serial interface, update the placeholder bytes and execute the code.

All ARM CPU exceptions are handled by U-Boot which allows us to classify the crashes accordingly. For example, the following exception is printed on the serial interface when the random bytes result in a illegal exception:

```
FREE undefined instruction
pc :          [<1ff50218>]      lr : [<1ff5020c>]
reloc pc : [<04016218>]      lr : [<0401620c>]
sp : 1eb19e68  ip : 0000000c   fp : 00000000
r10: 00000000  r9 : 1eb19ee8
r8 : 1c091c09  r7 : 1ff503fc   r6 : 1ff503fc
r5 : 00000000  r4 : 1ff50214   r3 : e0001000
r2 : 0000080a  r1 : 1ff50214   r0 : 00000005
Flags: nZCv  IRQs off  FIQs off  Mode SVC_32
Resetting CPU ...
```

The results of performing one thousand experiments are listed in Table 3.

The results show that executing random bytes in a NOP-like fashion is similar for bare-metal code compared to emulated and native user mode code. There seems to be less difference between Thumb and ARM but that could be due statistics.

Conclusion

Let us wonder no more. The results of this article tell us that the probability for executing random bytes in a NOP-like fashion for Thumb an ARM is significant enough to consider it a potentially relevant attack primitive. The probability is very similar for execution of emulated code, native user-mode code and bare-metal code. The number of random bytes executed impact the probability directly which matches our common sense. In Thumb mode, the density of bad instructions where the program crashes is higher than for ARM. One must realize the true probability for a given target cannot be determined in a generic fashion, thanks to memory mapping, access restrictions, and the surrounding code.

Type	4 bytes	8 bytes	16 bytes
NOP-like	32% / 52%	13% / 34%	4% / 13%
Illegal instruction	11% / 20%	14% / 29%	15% / 41%
Segmentation fault	52% / 23%	66% / 31%	73% / 40%
Unhandled CPU exception	1% / 2%	0% / 3%	0% / 4%
Unhandled ARM syscall	1% / 0%	1% / 1%	1% / 1%
Unhandled Syscall	1% / 1%	0% / 0%	0% / 0%
Unclassified	5% / 3%	6% / 2%	6% / 1%

Table 1. Probabilities for QEMU (Thumb / ARM)

Type	4 bytes	8 bytes	16 bytes
NOP-like	36% / 61%	13% / 39%	2% / 12%
Illegal instruction	13% / 19%	17% / 27%	23% / 40%
Segmentation fault	48% / 19%	66% / 33%	71% / 46%
Bus error	0% / 1%	0% / 1%	0% / 2%
Unclassified	3% / 0%	4% / 0%	4% / 0%

Table 2. Probabilities for native user (Thumb / ARM)

Type	4 bytes	8 bytes	16 bytes
NOP-like	53% / 63%	32% / 41%	7% / 19%
Undefined Instruction	16% / 20%	19% / 34%	25% / 51%
Data Abort	17% / 4%	25% / 7%	33% / 11%
Prefetch Abort	1% / 1%	1% / 1%	2% / 1%
Unclassified	15% / 12%	23% / 18%	33% / 18%

Table 3. Probabilities for native bare metal (Thumb / ARM)