Console Hacking 2006

Felix Domke tmbinc@elitedvb.net

November, 16th 2006

Abstract

The “Console Hacking 2006”-talk will present recent findings about the dominant gaming consoles, mostly regarding to their ability to run Linux and homebrew code. As two of the three consoles which the talk originally wanted to focus on are not yet available on the market, this paper will describe the difficulties in running own code on today’s consoles.

1 A small History of Gaming Console Security

When the Nintendo Entertainment System was released in USA in 1985, one of the few differences to the previously released “Nintendo Famicom” was the addition of a security chip inside the game cartridges. The chip, called ’CIC’ or ’10NES’, contains a small 4 bit microprocessor, which generates a pseudo-random sequence after poweron. The same chip, in a slightly different configuration, was inside the console, constantly comparing the locally generated pseudo random sequence with the stream received from the cartridge. When a difference was encountered, the processor reset would be activated. Without further modifications, this prevents simple ROM cartridges to be used for games, as they lack the output of the specific seuqence.

Of course this can be easily defeated, either by removing the embedded chip in the console [1] or by re-using an existing chip from a cartridge. However, cartridges without this chip could not be easily sold, as they would require a modification inside the console.

Later, Tengen, Atari’s NES games subsidiary, used a trick to get the sourcecode of the security chip from the USA copyright office [2], creating their own (clone) chip called “The Rabbit”. They went to court, Tengen lost.

Piracy wasn’t their only concern - Nintendo wanted to have control over the game market for their consoles, imposing strict guidelines to game publishers, and put down imports.

Even though the security concept was relatively simple, it worked out pretty well. Piracy cartridges often required usage of an “import adapter”, which was put in the console, and contained two slots - one for any original game, solely for the purpose of using the CIC, and another one for the pirated (or imported) game.

Hobbyist could run selfmade code on the NES, by taking an original cartridge and replace the ROMs with their own ROMs.

When cartridges were replaced by optical discs, notably by Sony’s PlayStation-console (launched in end of 1994 in Japan, and about a year later in USA and Europe), they couldn’t add a security chip to the games anymore, but the concept was the same: a hidden signature in the recorded track on disc lead to a magic sequence, transferred in a sideband (focus data) to the disc controller. The sequence was matched against a pre-recorded one. When it did not match, the game would not boot (the disc would be detected as an audio CD). The magic sequence was different for each of the three regions (Japan, USA, Europe), forming the ASCII-letters ’SCEI’ (for Sony Computer Entertainment, Inc), ’SCEE’ or ’SCEA’, depending on whether the game was made for the japanese, european or american market [6].

Again, the piracy protection was also used as an import protection. At some point, somebody managed to find the pin where the magic sequence was transmitted to the disc controller. An external chip, often called “modchip”, could be attached to overwrite the sequence with the correct one, thus allowing imports and copies to be played on modified consoles. As recordable medias became cheap, many people added modchips to their console for illegally playing copied games.

Homebrew code could also be simply burned to a CD and run in a modified console - the "magic sequence" was in no part depending on the actual game data, and there was no further protection.

Later consoles, like the Nintendo Gamecube, still use this type of protection, just in a more complex form. Instead of a simple "magic sequence", a more complex copy protection scheme was added, where the disc controller's firmware took care of authenticating a disc. When a disc was detected to be genuine, it's contents are trusted. Emulating the complete DVD-ROM is possible and has been done [5], and also allowed the injection of homebrew code into the emulated discs (though much easier solutions for executing homebrew code have been developed, which attack the way the system boots the bios, replacing the bios with a homebrew-friendly one which can load files from Network or SD card [7]).

All of these described systems can be attacked with a "man-in-the-middle"-attack, for example by doing the authentication with one source, then switching to another source for delivering the code and data. This can be done either on physical level (media swapping), or logical level (electronically muxing the data source, or use pre-recorded authentication information). This is great, because it allows us to execute our own code, which is the ultimate goal.

2 Today

Today, the situation unfortunately changed. Vendors want to keep their consoles secure, mainly to be able to sell premium content and keep their platforms controlled.

They use additional security to form a "chain of trust", so that only data which is known to be genuine is accepted for execution. This is usually done with public key cryptography.

At first, this is not stricly against piracy - after all, piracy involves playing back content which is made for being played back. It's against homebrew, imports, cheats and other modifications, in short: against any usage that is not intended. Yes, it's what we call DRM - the hardware, which you physically own, decides on itself which content it will accept, and which it won't.

The first generation of this type of security, most notably the original Xbox, relied on software only. It can be compared with a PC running an operating system, which simply doesn't allow you to execute binaries which are not "signed". Some additional hardware is necessary to prevent replacing the "BIOS", but that's all.

After this particular console has been hacked over and over [3], enabling running alternative operating systems, homebrew software like multimedia players and even pirated games from harddisk, vendors decided to invest more into security.

Without focusing on one specific implementation, it can be said that several technologies were, or are going to be, introduced into console systems. These technologies are not new - in fact, most of them are in use for several years on other systems, but they are reaching a new level for consumer electronics security.

The available techniques look pretty impressive at first sight:

2.1 Inventing a Secure Place

Gaming consoles are made to play games. Games want to use all the power of the system, after all, horse power directly translate to money required during the development and mass-production of a console. Engineers spent much time into tweaking the last bit of performance out of what was doable at the time of construction, and they don't want to waste any bit of performance for security.

Games rely on pushing data fast into the graphic subsystem, and nearly always use DMA for that. Now, making DMA access secure without slowing down the system is a tough task. A usual split of the whole system into a "supervisor" and "user space" doesn't work, as IO access is expensive from user space - it always has to go through the supervisor, so the 3D graphics libraries need to run in this context. Also, the game needs to run in the same context as their 3D libraries, in order to push data around without loosing speed.

Gaming consoles often use a "chain of trust" - every code which is loaded into the system should be checked to have a proper signature, and every code is in charge to not load any code without further verification. History shows that this doesn't work out. There is just too much complex software elements which are "in charge of being secure". Game programmers usually don't touch security issues at all (unless they are dealing with network code), and didn't invested much interest for example into verifying that a savegame hasn't been tampered with [8].

To remedy this unwinnable situation, something similar to a supervisor was required, just without the performance hit. An additional layer was introduced, a "hypervisor". While conceptually not too different to a "usermode" / "supervisor mode"-split, it allowed, with additional hardware support (as explained below), to build a secure subsystem inside the console environment. It's comparable to a "smart card" implemented in software. The hypervisor tries to do as less things as possible, in order to keep it clean, seperated and secure. A code bug in the hypervisor can compromise the whole system pretty easily, as nearly all security features are unter the control of the hypervisor. Extreme care must be taken to properly implement the hypervisor functions!

2.2 Non-Executable Pages

By default, memory which can be written is also executable. This behaviour makes it easy to execute own code for example in case of a stack overflow error.

By not allowing any executable page to be written outside of a fully trusted core (the hypervisor), there is no possibility to simply inject code in a buffer overflow exploit. Depending on the flawed code, it might be able to fill arbitarary registers with arbitrary values and jump to an arbitrary position in RAM, possibly with an arbitrary filles stack - but there is still no possibility to execute own code - unless of course if the trusted core, which has to be able to write code into memory, has a flaw.

Note that this also disallows the possibility to write self-modifying code, which might be a strong requirement for software-based CPU emulators. This can be partially workarounded by introducing sandboxes, in which writeable code pages are allowed, but no calls to important system functions. Code which might run there (through an exploit in the emulator software) then could still not execute important system functions, for example to communicate with the world other than with the given emulator functions.

It also puts down backdoors in software, which could otherwise be abused as "loaders". Without this a game developer could - knowingly or not - add a method to allow code to be loaded from an insecure medium into a normal game, submit this game, and get back a signed executable which runs on every console.

2.3 Encryption for Executable Pages

Code page attributes (like read-only) are solely based on the CPU's memory management unit. Writing to RAM can be done with a "DMA-attack", circumventing the CPU, thus circumventing the CPU's memory write protection. A "DMA-attack" ("direct memory access") can be done by either abusing well-defined interfaces to the RAM, or by creating them. Abusing well-defined interfaces can include Firewire [4], allegedly USB, Serial-ATA or PCI-express-man-in-the-middle-attacks. New interfaces can be created by interfacing memory chips directly, which is a quite complicated task, given that they run up to several hundreds of MHz in recent designs.

It's very hard to completely avert this type of attacks ("physical access wins"), so a good security system should not become insecure by "just" DMA attacks.

The reverse of DMA write attacks are snooping attacks, like Bunnie's successful original Xbox hack. Snoop attacks are also very hard to remedy, so the main attention is more and more just not to leave any important data unencrypted in ram.

2.4 Memory Checksumming

When memory is encrypted, and an attacker can write to the unencrypted memory, the attacker can not inject known blocks of data. However, by modifying data, he can overwrite a valid block with data which decrypts to "random". In certain situations, this can be used to gain an advantage for the attacker. For example, a system could have a revocation table, which stores hashes of known-bad (exploitable) executables. The attacker could overwrite the stored hash values with random (by storing any value to the unencrypted ram which doesn't decrypt with the - unknown - key to something useful). Also, changing the encryption key is usually an expensive task, so in order to gain performance, one might re-use keys more than once. If this is the case, memory regions can be "swapped". For example, a buffer can be read out by the attacker, and later copied back in, forming a replay attack.

The solution for these problems is memory checksumming. Memory checksumming calculates and updates checksums over memory and stores them in a secure place, for example in CPU on-chip RAM. When a checksum doesn't match, the memory was modified outside the CPU, and a security violation is detected and the system can be halted.

It requires additional memory for checksummed areas, but checksum blocks can be made of arbitrary size. It's a trade-off between memory and speed, as whenever a single byte of a checksummed block is accessed, the whole block must be read. However, when using cached memory, the time required to calculate the checksum can be hidden. Often it's enough to only protect really vital data, for example data belonging to the hypervisor.

2.5 Stack Canaries

A standard software feature is usage of stack "canaries" [9]. A special value, called canary, which must be random and different on each bootup, is stored globally. At the beginning of each function call, the canary value is stored on the stack, before the return address and saved registers. Before reading back register values at the end of the function call, the canary value on the stack is compared with the globally stored one. If they mismatch, the stack was overwritten, and an exception can be raised, possibly shutting down the system.

2.6 Bootrom inside CPU

When booting, most of the time a "chain of trust"-model is used: An initial bootloader checks the signature of the (potentially updateable) system kernel, and refuses to boot "unsigned" executables. It is vital that the bootrom can not be exchanged, as an atacker could replace the bootrom with one that does not check the signature. Also, often there is the attempt to have "security-by-obscurity" (which we all know doesn't work, but vendor's don't know this), so the kernel itself is encrypted. Sometimes, only encryption is used, in the hope that without the encryption key, an attacker can not inject useful code. In these cases, it's vital that the bootrom cannot even be read, as this would reveal the symmetric key which allows the attacker to encrypt their own kernel.

Thus, the bootrom is usually not a seperate chip, but embedded into another chip with bus access. For best security, the bootrom is placed directly into the CPU. However, this is often not as easy as it sounds. First, the costs for modifying a bootrom are extremely high, as changing the bootrom would require a new mask for the silicon process, which is easily in the multi-million dollar range. Flash and similar technologies have the problem that they are empty on production, and would require a "backdoor" to write them, and are often not available in the advanced CPU technologies. Also, they can become erased by electrical glitches.

2.7 RAM inside the CPU

Also important is that all data which is not encrypted cannot be considered as secret, and all data which is not signed (or at least checksummed with an internal secret) must be seen as untrusted. Sure, today's data busses are using speeds which are not easy to sniff or even to intercept, but basing a security on that is a very bad idea. At the time of the design, engineers must have had the tools to debug these busses, so why can't an attacker have the same tools? Yes, they are expensive. But because they are expensive, they are usually shared by more than one designer, so more people have access to them. Also don't forget that sometimes there is real money behind the attacker! They are able to pay professional design companies for building bus sniff devices.

Though still not inifinitely secure, busses inside chips are pretty secure. They can still be tapped, but at a much, much higher effort. While still not being a perfect solution - no consumer, high-speed CPU is really tamper-proof -, using CPU-internal storage can be considered as "pretty secure", or in other words: currently, there is just nothing better you can get.

However, they are still vulnerable against hardware glitches (like power spikes, clock glitches or radiation). As these types of hacks often require a lot of guessing, they can not be used "in field", but they can be used when a task has to be done a single time, for example to read out a bootrom.

Contrary to public belief, glitching does usually not require much information about a system. Nobody needs to know why exactly a clock glitch causes this and that behaviour, as that's nearly impossible to tell that without having access to the actual silicon implementation internals. It's enough to know that the behaviour changes. If the chance is 1:1000 that a glitch will invert the result of a signature check, that's no problem. Just keep it running a day or more. Glitch attacks are usually just not feasible for end-user hacks, as they don't want to wait a day for their console to bootup. Additionally, glitches often stress components very hard.

2.8 Fuses

Using electrical one-way programmable bits is nothing new in chips. They provide an alternative to eeprom or flash cells, which often cannot be implemented in the same silicon technology on one die.

Fuses can be seen as a small one-time-programmable space inside the CPU, and can be used to contain keys or a serial number, to "pair" a ROM image to a CPU. Additionally, it can be used for configuration information. For example, a specific (signed, thus unpatchable) Flash-ROM image can decide to refuse to boot when the configuration, or serial number, stored in CPU does not match the ROM image. It can also be used to lock out software versions which are known to contain bugs, like a revocation list.

3 Summary

Still, the weakest part dominates the whole system. Whenever all stored information of a single hardware component are known, it can be emulated. On some platforms this can be done with the DVD-ROM, which often does not contain as much security as the rest, for whatever reasons. It does not allow homebrew software to be run (the data coming from the drive is still considering 'untrusted' until it's sign-checked), but it does allow copying medias, without touching the (hard) security at all. Of course this doesn't help us. We don't want to play copies, but run our own software.

Console security is no funny thing anymore, it has become something real. Still, that's no reason to despair. A single error in an important piece might be enough, and even though vendors have learned from their faults, nobody is perfect. Maybe someday vendors will learn that allowing to run real, free operating systems on their platform is in fact good marketing and overall a good thing?

References

[1] Disabling the NES "Lockout Chip", Mark K., `http://nesdev.parodius.com/nlockout.txt`

[2] Lawsuit: ATARI GAMES CORP. and TENGEN, INC. (Plantiff) NINTENDO OF AMERICA INC. AND NINTENDO CO., LTD., (Defendant) - Security Code `http://www.nesplayer.com/features/lawsuits/tengen.htm`

[3] 17 Mistakes Microsoft Made in the Xbox Security System, Michael Steil, `http://www.xbox-linux.org/wiki/17_Mistakes_Microsoft_Made_in_the_Xbox_Security_System`

[4] Hit by a Bus: Physical Access Attacks with Firewire, Adam Boileau, `http://www.ruxcon.org.au/files/2006/firewire_attacks.pdf`

[5] `http://www.crazynation.org/GC/Interface.htm`

[6] `http://club.cdfreaks.com/showthread.php?t=48477`

[7] `http://www.gamecubeos.com/modules/news/`

[8] Technical Analysis of 007: Agent Under Fire save game hack, Anonymous, `http://xbox-linux.sourceforge.net/docs/007analysis.html`

[9] Windows Stack Buffer Overflow Protection, Jason Coombs, `http://www.ddj.com/184405546`