History of Hacking the Nintendo 3DS

Shavina Chau Joonho Ko Jessica Tang
{shavinac, joonhok, jynnie}@mit.edu

INTRODUCTION

In March 2011, the Nintendo 3DS hit worldwide markets. Also known as the 3DS, Nintendo's newest handheld gaming console included stronger security than all its preceding consoles. But the 3DS was completely hacked by a community of users including yellows8, derrek, pluto, and smealum.

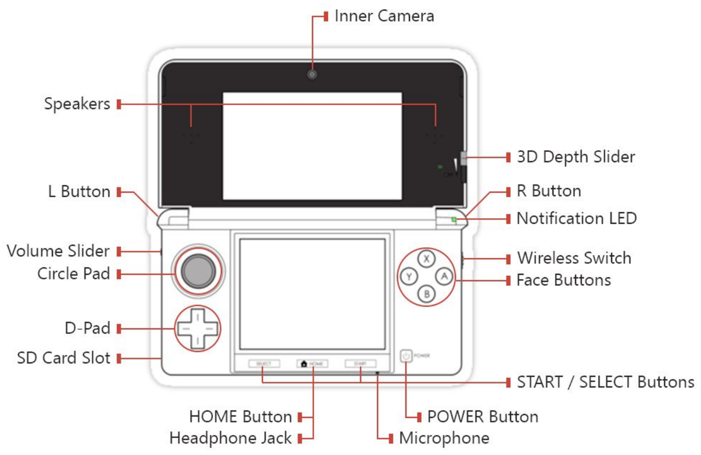

Figure 1: Diagram of the Nintendo 3DS [1]

Console hacking has long been fueled by users who want more control of their console than provided out of the box. The common belief of why hackers hack consoles is to pirate games. However, the community effort behind breaking the 3DS and many other gaming consoles was motivated by homebrew. Homebrew is unlicensed software, including games, produced by consumers for proprietary platforms. 3dbrew defines homebrew as "a popular term used for applications that are created and executed on a video game console by hackers, programmers, developers, and consumers" [2]. Many consoles like the 3DS have rigorous security measures in place that prevent users from executing unofficial, custom software and applications. At the same time, to officially develop for Nintendo consoles, one has to purchase the Nintendo software developer kit and pass through stringent screening processes. As such, official routes of development are often expensive and gated for consumers to produce their own applications. Instead, communities tackle hacking consoles.

Our paper will be outlining the various exploits and hacks applied to the 3DS. We'll first outline and analyze the 3DS infrastructure and security measurements. We then dive into the different hacks that gave users control of each privilege level. Finally, we discuss the improvements made by hackers on the base hacks we discussed and Nintendo's attempts to patch vulnerabilities.

3DS INFRASTRUCTURE AND PRIVILEGE LEVELS

Before we begin dissecting how the 3DS was hacked, we will begin by discussing the infrastructure of the 3DS and its inferred security policies.

The 3DS has three privilege levels for users: ARM11 user mode (also referred to as userland), ARM11 kernel mode, and ARM9 kernel mode (see Figure 2). ARM11 user mode is the most shallow privilege and is the privilege most processes have access to, such as games, browsers, and the home menu. A process with ARM11 kernel mode access also has ARM11 user mode access, but not ARM9 kernel mode access. These levels are defined by the processor (which ARM) used and mode (user or kernel mode) run. Each processor has access to specific resources and each mode determines to what level of access a process has to a processor's given resources.

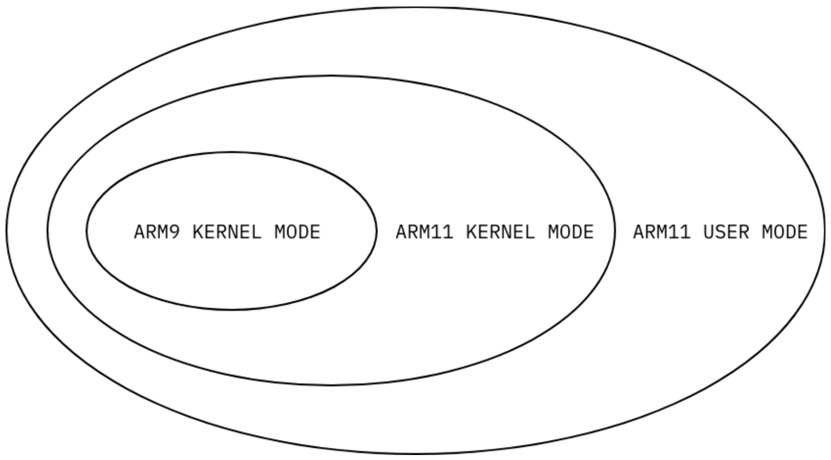

Figure 2: Privilege Levels depicted as a shell

User mode is a mode where processes have limited access to that processor's resources. Kernel mode grants a process full access to that processor's system resources: including memory addresses.

The processors used by the 3DS are ARMs, Advanced RISC Machines [3]. RISC machines are Reduced Instruction Set Computers. ARM11 is the primary processor that handles most tasks, including the Operating System. ARM9, on the other hand, runs a single process called process9 on user mode and is otherwise responsible for lower level tasks and cryptographic functions such as encryption and signature checks. In the 3DS infrastructure, the ARM11 and ARM9 have access to different resources to ensure the separation of concerns. The ARM11 and ARM9 share access to main memory resources (WRAM, VRAM, and FCRAM), while the ARM9 maintains an internal memory with its own code and data that the ARM11 does not have access to. The ARM9 also exclusively has access to the keyscrambler and AES engine, the critical cryptographic functions of the 3DS. See Figure 3 for a more in depth diagram of system resource allocation (bolded modules will be discussed in later sections).

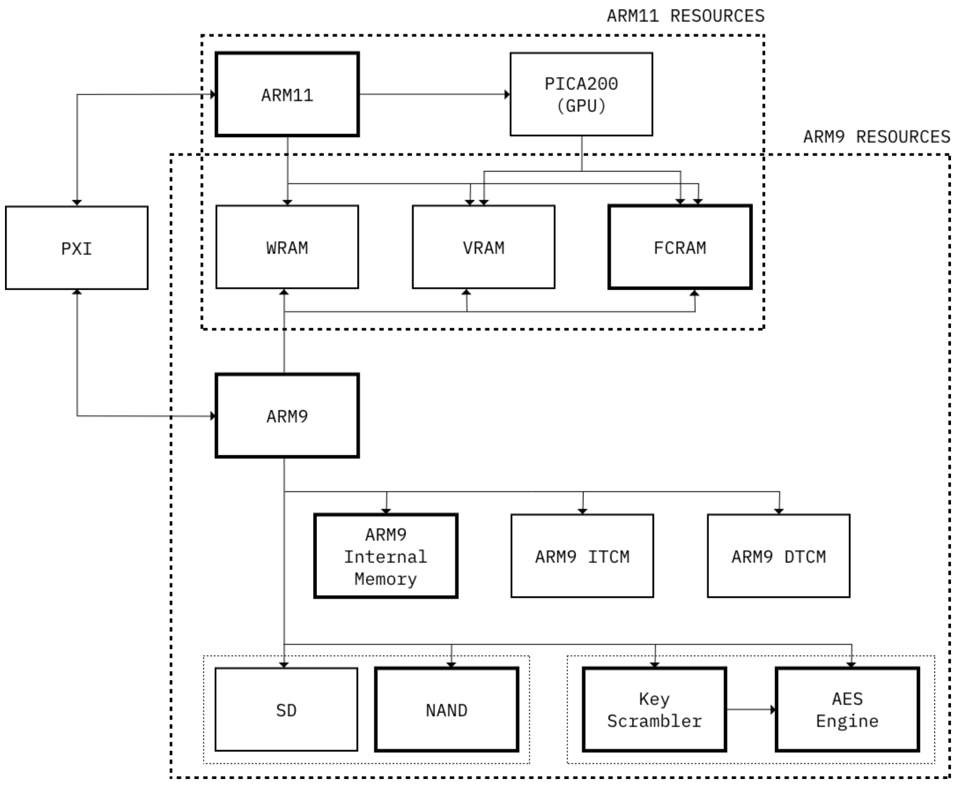

Figure 3: 3DS Infrastructure and ARM Resources

Separating the resources each processor has access to makes it harder for adversaries to gain complete control over the 3DS. For example, exploiting functionality of common user facing functions, such as running a game, does not affect the security of cryptographic functions. Thus Nintendo strongly implements privilege levels on a hardware level by limiting resource access and on a software level through user and kernel modes.

We'll now discuss some specific security protocols at different privilege levels.

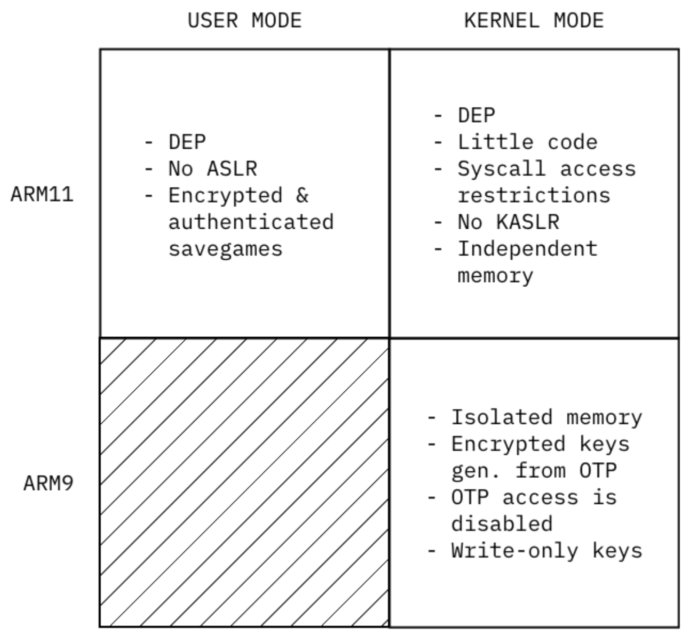

Figure 4: Specific privilege level security

ARM11 user mode and kernel mode are strictly protected by Data Execution Prevention (DEP). DEP essentially checks parts of memory marked non-executable to prevent executing code in those locations (including heaps) [4]. This makes code injection much harder for adversaries. ARM11 user mode encrypts and authenticates savegames so they are not great initial entrypoints for adversaries.

In addition, ARM11 user mode prevents access to certain parts of memory. In FCRAM, processes have access to application data and code and part of system memory. For example, it does not have access to Nintendo's operating system (NS) code and data, which is part of system memory. See Figure 5 for more details of what ARM11 user mode has access to in FCRAM.

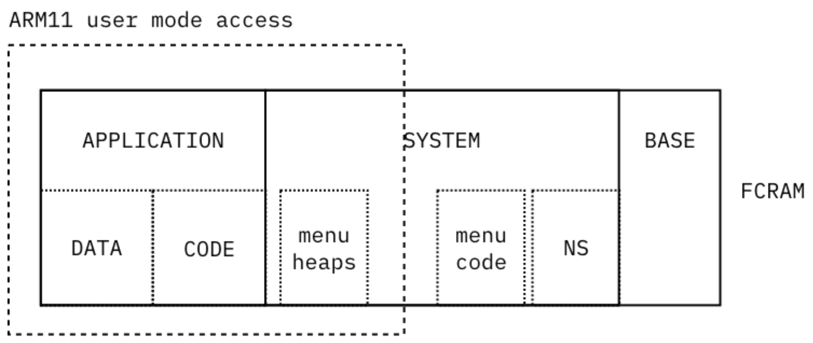

Figure 5: Diagram of ARM11 user mode's limited access to FCRAM

A key vulnerability in both ARM11 user and kernel mode is that neither use address space layout randomization (ASLR or KASLR for kernel ASLR). ASLR randomly rearranges system memory so that adversaries cannot reliably jump to a certain address in memory to access data and functions in memory. While there is no KASLR for ARM11 kernel mode, Nintendo has been changing the memory addresses used with every kernel update – perhaps recognizing the vulnerability.

While the ARM9 is meant to be the security processor, it has rather lax security. It is backwards compatible with the Nintendo DS, meaning vulnerabilities with the ARM9 of the DS are exploitable in the 3DS. Any .data and .stack files are executable, which would allow adversaries to execute code there so long as they could jump to the correct memory address. And .text files are writable.

The chain of trust for the ARM9 does provide some security. When the bootrom for the ARM9 is loaded, it's code is verified before jumping to firmware binary for execution. In addition, a cryptographic layer (called the ARM9Loader) that's part of the firmware binary protects the ARM9 kernel. The ARM9Loader loads new keys generated from an encrypted per-console key created from a one-time pad (OTP). The OTP is, as it should be, only used once and disabled after use.

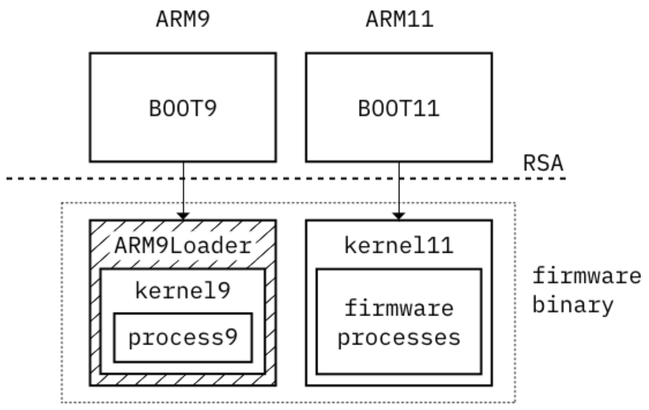

Figure 6: Chain of trust

The AES engine is the hardware cryptographic engine that is used for all block ciphers by the 3DS. For security, it ensures keys are write-only, so an adversary can't simply read the keys to gain access. And also uses a keyscrambler so keys are calculated by hardware and never exposed to the CPU.

Altogether, based on the security protocols and privilege levels we've examined, we've written a brief security policy for the Nintendo 3DS:

> Definitions: The **Nintendo 3DS** (also referred to as the 3DS) is a video game console for Nintendo 3DS and Nintendo DS games. Games and applications are either loaded from an official cartridge or downloaded off the Nintendo eShop. Users can connect to the internet with the 3DS to facilitate processes like accessing the Nintendo eShop and playing online multiplayer games.
>
> Roles: Roles are defined to help outline privilege levels, as such, they may include definitions of users, groups, and processes.

- **USER** – users include players and game developers accessing the video game console. They should be able to play, download, and save official 3DS or DS games and applications. They should additionally be able to access menu settings and a home menu of previously downloaded games and applications.
- **GAME PROCESS** – games should be able to load and run their code; save data onto the console; and access previously saved data. Games should also be able to connect to the internet and access other peripheral inputs, such as inputs from the D-pad and touchscreen, and outputs, such as the two screens.
- **CONSOLE PROCESS** – base processes including the Nintendo 3DS operating system (known as NS) and menu process should be able to load and run their code; save data; and modify access peripheral inputs and outputs.
- **NINTENDO** – the console developer and company should be able to build the firmware, push updates to console processes, and push updates to cryptographic functions. Updates pushed by Nintendo should be downloadable to any 3DS connected to the internet.

Security Goals:

- USERs, GAME PROCESSes, and CONSOLE PROCESSes should not have access to critical cryptographic functions, including key generation.
- USERs and GAME PROCESSes should not be able to extract, read, or write Nintendo's operating system code and data.
- USERs should not be able to extract, read, or write code from GAME PROCESSes off the 3DS (DRM Content Protection) [5].
- All processes running on the system should be authenticated by Nintendo [5].
- All data that can be encrypted should be encrypted.
- 3rd party code not loaded from an official game cartridge or downloaded from the Nintendo eShop should not be executable.
- Integrity of savegames should be preserved.
- Cryptographic keys should be kept confidential and integrity preserved.

In order for homebrew to be executable on the 3DS, hackers will need to gain ARM9 kernel mode privilege. The hacks we detail in the following sections will sequentially gain access to every level of privilege and break the above security goals.

CRACKING THE ARM11

We first get a foothold into the system by tackling the first privilege layer, the ARM11 user mode. Several exploits exist that achieve custom code execution on the ARM11 user mode. These exploits are influenced by the existence of Data Execution Prevention (DEP), which marks pages of memory as non-executable, preventing code to be run from locations in memory where it should not. Much of the overview for breaking into the ARM11 user mode was explained by Smealum during the 2015 32C3 conference [6].

ROP

ROP, which stands for Return Oriented Programming, is a technique where a fake stack can be built to point to code snippets just before return instructions. By doing so, attackers can execute code using preexisting system instructions and manipulating return addresses instead of actually injecting their own custom code.

The downside to ROP is that it is quite limited in what it can do, since it can only be used to gain access to preexisting instructions, and it is annoying to actually implement. In addition, in the case of the 3DS, one still does not have access to any system calls that allow mapping of executable pages in memory. ROP is still useful, however, as it will be used in the next section to gain access to many capabilities that will serve as a foundation for finally running homebrew.

GSPWN

A technique used in conjunction with ROP is called GSPWN, which exploits the GPU's DMA (Direct Memory Access) to FCRAM to override a currently running application to achieve custom code execution while bypassing DEP. By rendering data into main memory which contains application code, GSPWN can overwrite an application's .text file.

Unfortunately, although this bypasses DEP, we are still limited to the current application's sandbox. This means that although we can manipulate an application's savedata, things like the SD which live outside the application are still out of reach. However, the GPU curiously has access to most of the memory region called the 'SYSTEM region' (Figure 5) which includes the 3DS home menu, internet browser, and a system module called 'NS' (Nintendo Shell). Although the menu code and NS code are outside the range of the GPU, the menu heap is well inside this range. Using the menu heap, it is possible to achieve ROP execution of the home menu.

With ROP access of the home menu, we gain access to several new things, including a preexisting system call, called NS:S, that allows creation or death of any process, access to the SD card, decryption of any system title or game, and access to any extdata, extra data stored on the SD card or NAND. It is these new capabilities that are used as the base for running homebrew.

Running Homebrew

A homebrew process, written with ROP, is implemented as a background 'service' under the menu:

1. The current application is killed using NS:S.
2. A new application is opened using NS:S with the necessary permissions for target homebrew.
3. The application is taken over using GSPWN.

4. The homebrew service sends handles to the application (for SD access, etc.)

We now have the capability to run code under any ARM11 user mode application. However, each time we wish to run this code we have to redo this entire process, which is quite time-consuming. In addition, we are still limited by the privileges of the ARM11 user mode, i.e. we don't have access to the ARM11 kernel or ARM9.

Secondary Entrypoints

With these privileges, however, we can use any application or game as a secondary entrypoint. There are countless entrypoints that have been discovered by the 3DS homebrew community: several of the more well-known ones will be mentioned below. In general, the name of an entrypoint is given the name *[name]-hax*, where *[name]* is the medium or application used.

Menuhax exploits faulty theme-handling code in the 3DS menu on startup. This allows homebrew to be run on boot time, which is extremely convenient and making menuhax one of the more common ways of starting homebrew.

Cubic Ninja is a cute action-adventure puzzle game that has a level-sharing feature that allows users to share QR codes to play other peoples' levels. *Ninjhax* exploits the QR feature to allow a malformed QR code to be downloaded, triggering the execution of homebrew. As a humorous aside, *Cubic Ninja* became extremely popular due to *Ninjhax* and sold out at major video game retailers, causing online retailers to sell *Cubic Ninja* as high as $500 at one point in time, despite its mediocre gameplay rating.

SNSHAX

Using GSPWN, the GPU can only access a fraction of the SYSTEM memory region, which means the GPU cannot access NS (Figure 5). Being able to GSPWN NS would be useful for several reasons. NS allows for a way to downgrade individual titles, allowing users to revert software to an older version pre-patch to use a secondary entrypoint exploit. NS also has access to several useful system module-specific system calls.

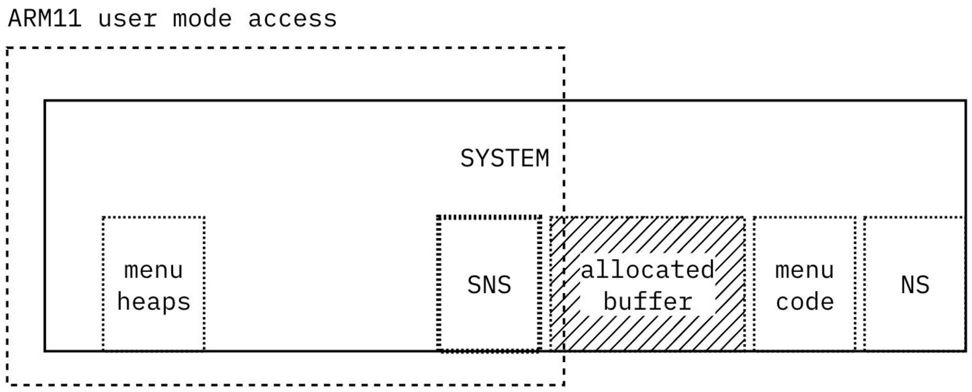

Figure 7 (SNSHAX): Allocating a buffer to push SNS into an accessible region for GSPWN.

Hypothetically, it *would* be possible to access the NS region if it was possible to kill the instance of NS, allocate a buffer to take its place, and then start a new instance of NS, which is pushed down into a lower region of SYSTEM that can be accessed by the GPU. Unfortunately, this isn't so simple: NS needs to be running in the first place to launch NS, and you cannot launch multiple instances of the same title. Luckily, the 3DS has a "Safe Mode" version of NS (SNS) that has a different title ID, which is functionally equivalent to NS. Therefore, a buffer can be allocated while keeping the original NS instance, and starting a new SNS instance that can be GSPWNed (Figure 7).

Cracking the ARM11 Kernel

The above gave us ARM11 user mode privileges. In this section, we give a high-level overview of Memchunkhax, which is the most common method of access to the next-highest privilege level, the ARM11 kernel. (As an aside, the ARM11 kernel is the first game console kernel developed by Nintendo: previous consoles' "kernels" were run directly on hardware.)

Memchunkhax exploits the fact that the kernel memory reads from a structure called the *memchunk header* even after it has been mapped to userspace. This kernel structure lives inside the FCRAM, which is within the GPU DMA region (GSPWN). Therefore, Memchunkhax can overwrite the memchunk pointers in FCRAM to gain write access to the kernel's pages by mapping them to userspace.

Memchunkhax v1 was patched in 2014 by Nintendo: the new kernel version now verifies every memchunk header to see that is has not been malformed. An updated version, Memchunkhax v2, still overwrites memchunk pointers as before, but uses a slightly more complicated process to do so, involving the kerner's slab heap structure, which will not be discussed.

Gaining ARM11 kernel mode privilege gives us access to direct address memory and the full resources of ARM11, thus broadening the kinds of homebrew that can be run.

Summary

The ROP, GSPWN, and SNSHAX techniques show that even while restricted to the ARM11 user mode, it is possible to get fairly high levels of privilege on the 3DS. Giving away access to physical memory is dangerous and potentially easily exploitable, as can be seen through GSPWN.

CRACKING THE ARM9

Even greater privileges are necessary to install more extensive homebrew such as custom firmware, as well as gain access to encryption/decryption keys for system data. The next goal for the 3DS hacking scene was to break into the ARM9 processor and gain earlier control of the system. The ARM11 and ARM9 processors share certain resources, including sections of shared memory; if we can gain full control over the ARM11 as described previously, we may also find a way to take control of ARM9 by exploiting the shared space. We detail three such hacks: NTRCardHax, ARM9LoaderHax, and Boot9strap.

NTRCardHax

One such exploit is known as "NTRCardHax," which requires a modified DS cartridge. The ARM9 processor, in addition to serving as a security processor, is also used for backwards compatibility with DS games. The interface for reading a DS cartridge resides in the ARM11/ARM9 shared IO region, which is a vulnerability since the ARM9 is the only processor using the region yet the ARM11 still has access to it.

Normally, the process for reading in a DS cart is as follows:

- Start by writing to the CTRL register, specifying a transfer size of 0x200 bytes.
- Begin a read loop: wait for room in the buffer, read in the next 0x200 bytes into the buffer.
- Repeat the read loop until finished.

However, we can overwrite the CTRL register using the ARM11, since it is in the shared IO space between the ARM11 and ARM9. Specifying a transfer of 0x4000 bytes instead causes a buffer overrun. This means we can overwrite additional space in memory with data read in from the DS cart; with a customizable or modified DS cartridge, we can now execute our own code on the ARM9 processor [6].

ARM9LoaderHax

A different exploit, known as "ARM9LoaderHax," is another way to gain control over the ARM9. As described in Section I, the firmware binary for the ARM9 kernel is wrapped inside an additional cryptographic layer, ARM9Loader, which generates new keys based on the console's unique OTP.

The original implementation of ARM9Loader is as follows:

1. Calculate the SHA256 hash of OTP.

2. Read the key-sector data from NAND flash memory.
3. Decrypt the first key using the OTP hash and put it in keyslot 0x11.
4. Use the 0x11-key to generate a bunch of keys.
5. Verify the 0x11-key by encrypting a fixed test-vector.
6. Decrypt ARM9 binary.
7. Jump to entrypoint.

However, one major flaw was that the 0x11 keyslot is never cleared - this means that with access to ARM9 code execution, all keys can simply be regenerated using the 0x11 keyslot. Once a keyslot is in use by software to store data, the keyslot's initialization vectors (e.g., KeyX, KeyY) cannot easily be changed without risk of breaking software that relied upon that keyslot. However, it is possible to generate a different initialization value for a (currently unused) keyslot, and depend on that new value, which is what Nintendo did [6].

After this exploit was discovered, Nintendo released a new firmware patch that updated the ARM9Loader implementation and uses a different key from NAND. That implementation is as follows:

1. Calculate SHA256 hash of OTP.
2. Read the key-sector from NAND.
3. Generate all the previous keys, for compatibility.
4. Decrypt key #2 from NAND.
5. Decrypt ARM9 binary using key #2.
6. Clear keyslot 0x11.
7. Jump to entrypoint.

While the keyslot is now cleared, another major flaw exists here. Key #2 is not verified by a test-vector. This is the basis for ARM9LoaderHax, which requires writing to NAND memory. If we supply our own key #2, the ARM9 binary will decrypt to garbage but will still jump to the entrypoint - with enough trial-and-error with key #2, we eventually find some garbage that decodes to a branch instruction. With more trial-and-error, the branch instruction will jump to a location outside of the ARM9 binary - this is where we put our payload and allow us to execute arbitrary code from the ARM9 processor [6].

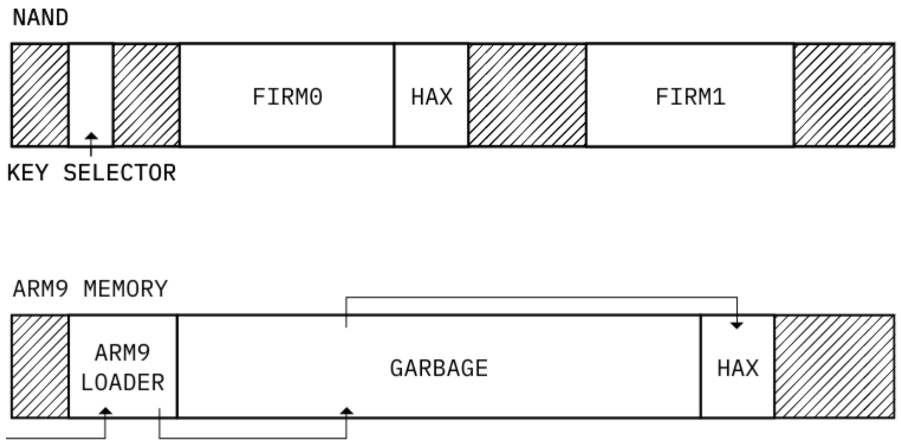

Figure 8: ARM9Loader exploit to reach payload

How do we get the payload onto the protected ARM9 memory? We exploit a few different aspects of the system. First, we can overwrite data in NAND memory, which stores the key-sector as well as two firmware partitions, firm0 and firm1. The partitions contain code for the ARM9 firmware binary, and is encrypted with the console's (correct) key #2. Next, the bootrom does not clear the memory used by firm0 when it fails to be validated - this is important in keeping our payload in memory [6]. In addition, firm0 is larger than firm1; firm1 acts as a backup in case firm0 becomes corrupted. To install the payload into ARM9 memory, we first put the payload inside the end of firm0. Upon rebooting the console, the ARM9 bootrom is executed and loads firm0 into ARM9 memory (with our payload at the end). It will attempt to decrypt the binary, but fails a hash-check because the binary is modified with our payload. Because of the failure, the bootrom will instead load firm1 on top - remember that firm1 is smaller, so our payload remains in memory. Since the firm1 binary is valid and passes the hash-check (we did not modify it at all), ARM9Loader will begin execution, decrypting the binary to garbage (remember we overwrote the keyslot). As described in the previous paragraph, we hope that the garbage decodes to a branch instruction that will jump to the payload and execute the code there. Thus, we now have the ability of ARM9 code execution from a cold boot early in the process [6].

How feasible is this exploit? In other words, how much trial-and-error is needed to successfully decode the branch instruction that jumps to payload code? Once a working key is found, it can be reused on the same console. In addition, we can make as many attempts as we want. Upon failure, we only need to rewrite the NAND key-sector with a new key to try, and reboot the console again. One estimate gives an approximately 23% chance of success on each try, based on these probabilities [7]:

- The probability of an exception (e.g. invalid memory dereference, invalid instruction) for any randomly-generated instruction is relatively close to 1/8. Thus, each boot will execute (on average) eight random instructions before throwing an exception.
- The probability of one of those eight random instructions being a BRANCH instruction is about one in eight (bits 27-25 == 101b), of which ~1/2 will have the right CONDition bits

set, presuming randomness. Thus, a rough estimate is a 50% chance that a BRANCH instruction will execute for any random decryption key.

- So, there is approximately a 50% chance (per boot with random decryption key) of a valid BRANCH instruction being executed. Some portion will jump to another location that's filled with cryptographically random instructions (within firm1's size). How big is the firmware image in memory? 50% * (1 - (sizeof firmware)/256MB) == chance of hitting memory left over from prior boot, and running the exploit code.
- The ARM9 partition in 9.5 NATIVE_FIRM is 566784 bytes. This results in: 50% * (1 - (566784 / 1048576)) = .5 * (1 - .5405) = .5 * .46 = 23% chance.

Thus, the ARM9LoaderHax exploit is quite feasible, depending on how difficult the task of rewriting data to NAND is.

AES Engine

The ARM9 is also responsible for a hardware AES engine, which is used heavily for purposes like encrypting and decrypting installed content titles (e.g. games). The AES engine had two security features: write-only key slots (the key cannot be read back), and keyscrambling. The actual key used is calculated in hardware and never exposed to the CPU. The actual key is derived as a function of two 128-bit keys: normal_key = F(keyX, keyY), where F is implemented in hardware. However, the keyscrambler function was discovered through differential cryptoanalysis and leaked key information [6]. With (keyY, normal_key) pairs leaked from shared communication capabilities between the 3DS and the WiiU (another Nintendo console), as well as from 3DS firmware images, it was possible to also extract keyX using flaws in the weak keyscrambler function [5]. With the keyscrambler function fully known, it was now possible to perform encryption/decryption outside of the 3DS system - for example, to decrypt game save files on a computer.

Boot9strap

The next significant development in 3DS hacking was the discovery of an exploit known as “sighax,” further developed in a later exploit called “Boot9strap.” This exploit enables users to gain control even earlier than ARM9LoaderHax, and allow the system to boot from a custom firmware image stored on the SD card. The paper “Attacking the Nintendo 3DS Boot ROMs” by Michael Scire et. al. describes the pertinent system details and flaws, the implementation details of their attack, and their results. First, they describe “bootchain” - the chain of trust in the 3DS system’s boot process. The root of trust for 3DS security is within the boot ROM (Read Only Memory), which is burned into the processors during manufacture and cannot be changed. The ARM9 boot ROM (Boot9) contains RSA public keys (only Nintendo has the matching private keys), which used to ensure that only Nintendo-signed firmware images will run on the system. To improve security of the bootchain, both Boot9 and its ARM11 counterpart (“Boot11”) are split into two halves. The first “unprotected” half is readable from firmware given sufficient privileges, while the second “protected” half is made unreadable early in the boot process [8].

The authors then outline the steps in the 3DS boot process as follows:

1. Initialize AES keyslots with secret keys from Boot9 (using AES hardware to decrypt all subsequent data read from boot device)
2. Initialize RSA keyslots with firmware public keys from Boot9 (using separate keys for various signature and console types)
3. Select boot device (typically NAND flash storage)
4. Read firmware header from selected boot device to memory
5. Validate SHA-256 hash and RSA-2048 signature of firmware header
6. Read firmware sections to memory according to parameters specified by firmware section headers
7. Disable access to protected halves of Boot9 and Boot11
8. Jump to ARM9 and ARM11 entrypoints specified by firmware header

Scire et. al. note that this process seems conceptually secure, but the flaws are found in the implementation. Specifically, the authors discovered multiple flaws in Nintendo's signature parser; normally, the firmware header's signature is verified by calculating the hash of the data, decrypting the signature using the public key (found in Boot9), then verifying that the calculated hash matches the hash embedded in the signature. To determine the proper offset of the location of the embedded hash, the signature parser relies on the signature's specified length values. However, because the signature parser does not bound check these values (allowing them to point beyond the signature block), an exploit which modifies the length values can cause the signature parser to use an incorrect offset and verify against the memory area storing the calculated hash value. In other words, when the signature parser attempts to verify the calculated hash against the "correct" embedded hash value, it will instead verify the calculated hash value against itself, which always succeeds. This means any firmware image using this signature will appear to be validly signed to the signature parser, enabling the "fakesigning" of arbitrary firmware images [8]. In further analysis, the authors were able to determine exactly which length values were improperly verified as well as the exact stack location of the calculated hash value. With a custom fakesigned firmware image, the authors were able to execute code that dumps the protected halves of Boot9 and Boot11, and have the system load a second custom firmware image from the SD card and continue the boot process [8]. "Sighax" is the ability to "fakesign" custom firmware images, and "Boot9strap" relies on this exploit to gain reliable Boot9/Boot11 code execution through a custom firmware implementation [9].

NTRBoot and Flashcarts

After examining the data from the dumped protected half of Boot9, Scire et. al. discovered an alternative boot method. Before Boot9 attempts to load a firmware image from NAND, it checks to see if the device's shell is closed and if a specific key combination (START+SELECT+X) is being held - if so, Boot9 will attempt to load a signed firmware image from an inserted standard DS cartridge ("NTR" cartridge). By utilizing an exploit RSA signature, it is possible to boot a custom firmware image from one of the many commonly available rewritable NTR cartridges (also known as "flashcarts"). Using a similar implementation as before, this enables code execution to install a custom firmware image to NAND flash storage. While normally it would be impossible to perform the required key presses while the device's shell is closed, the shell sensor can be tricked using a magnet [8].

CONCLUSIONS

At this point in time the 3DS is considered completely hacked. Fewer patches have come out from Nintendo as they focus on their newer consoles. However, from analyzing the various DS hacks, we can conclude that in order for consoles to be more secure, security features should include: random assignment of physical memory addresses (ASLR), not sharing resources - specifically memory - between different processors, ensuring calls on RISC machines are within proper bounds, and keeping the integrity and confidentiality of secret keys.

RESOURCES

[1] Arkhandar. "Regular Nintendo 3DS button and features layout". Wikipedia, 2013. https://en.wikipedia.org/wiki/Nintendo_3DS#/media/File:Nintendo_3DS_Button_Map.png

[2] 3DBrew. "3DBrew: a wiki dedicated to homebrew on the Nintendo 3DS". 2015. https://www.3dbrew.org/wiki/Main_Page

[3] WikiTemp. "3DS Glossary". GBAtemp.net, 2017. https://wiki.gbatemp.net/wiki/3DS_Glossary

[4] Microsoft Support. "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003". Microsoft, 2017. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in

[5] Lu, Yifan. "The 3DS Cryptosystem". 2016. https://yifan.lu/2016/04/06/the-3ds-cryptosystem/

[6] derrek, plutoo, smealum. "Breaking the 3DS". in 32c3 Proceedings, Germany, 2015. https://smealum.github.io/3ds/32c3/#/

[7] Selver. "Arm9Loader v2". GBAtemp.net, 2015. https://gbatemp.net/threads/arm9loader-technical-details-and-discussion.408537/#post-5934450

[8] Scire, Michael, et. al. "Attacking the Nintendo 3DS Boot ROMs". arXiv. 2018. https://arxiv.org/pdf/1802.00359.pdf

[9] SciresM. "sighax and boot9strap: Presentation by SciresM". in 33.5c3 Proceedings, 2016. https://sciresm.github.io/33-and-a-half-c3/