17 Mistakes Microsoft Made in the Xbox Security System

Michael Steil <mist@c64.org>
Xbox Linux Project http://www.xbox-linux.org/

Introduction

The Xbox is a gaming console, which has been introduced by Microsoft Corporation in late 2001 and competed with the Sony Playstation 2 and the Nintendo GameCube. Microsoft wanted to prevent the Xbox from being used with copied games, unofficial applications and alternative operating systems, and therefore designed and implemented a security system for this purpose.

This article is about the security system of the Xbox and the mistakes Microsoft made. It will not explain basic concepts like buffer exploits, and it will not explain how to construct an effective security system, but it will explain how *not* to do it: This article is about how easy it is to make terrible mistakes and how easily people seem to overestimate their skills. So this article is also about how to avoid the most common mistakes.

For every security concept, this article will first explain the design from Microsoft's perspective, and then describe the hackers' efforts to break the security. If the reader finds the mistakes in the design, this proves that Microsoft has weak developers. If, on the other hand, the reader doesn't find the mistakes, this proves that constructing a security system is indeed hard.

The Xbox Hardware

Because Microsoft had a very tight time frame for the development of the Xbox, they used off-the-shelf PC hardware and their Windows and DirectX technologies as the basis of the console. The Xbox consists of a Pentium III Celeron mobile 733 MHz CPU, 64 MB of RAM, a GeForce 3 MX with TV out, a 10 GB IDE hard disk, an IDE DVD drive, Fast Ethernet, as well as USB for the gamepads. It runs a simplified Windows 2000 kernel, and the games include adapted versions of Win32, libc and DirectX statically linked to them.

Although this sounds a lot more like a PC than, for example, a GameCube with its PowerPC processor, custom optical drive and custom gamepad connectors, it is important to point out that, from a hardware point of view, the Xbox shares *all* properties of a PC: It has LPC, PCI and AGP busses, it has IDE drives, it has a Northbridge and a Southbridge, and it includes all the legacy PC features such as the "PIC" interrupt controller, the "PIT" timer and the A20 gate. nVidia sold a slightly modified Southbridge and a Northbridge with a another graphics core embedded for the PC market as the "nForce" chipset between 2001 and 2002.

Motivation for the Security System

The Xbox being a PC, it should be trivial to install Linux on it in order to have a cheap and, for that time, powerful PC. Even today, a small and silent 733 MHz PC with TV connectivity for 149 USD/EUR is still attractive. But this is not the only thing Microsoft wanted to prevent. There are three uses that should not have been possible:

- **Linux**: The hardware is subsidized and money is gained with the games, therefore people should not be able to buy an Xbox without the intent to buy any games. Microsoft apparently feels that allowing the Xbox to be used as a (Linux) computer would be too expensive for them.
- **Homebrew/Unlicensed**: Microsoft wants the software monopoly on the Xbox platform. Nobody should be able to publish unlicensed software, because Microsoft wants to gain money with the games to amortize the hardware losses, and because they do not want anyone to release non-Internet Explorer browsers and non-Windows Media Player multimedia software.
- **Copies**: Obviously it is important to Microsoft that it is not possible to run copied games on the Xbox.

Microsoft decided to design a single security system that was supposed to make Linux, homebrew/unlicensed software and copies impossible. The idea to accomplish this was by simply locking out all software that is either not on the intended (original) medium or not by Microsoft.

On the one hand, this idea makes the security system easier and there are less possible points off attack. But on the other hand, 3 times more attackers have a single security system to hack: Although Open Source and Linux people, homebrew developers, game companies as well as crackers have little common interests, they could unite in this case and jointly hack the Xbox security system.

Of the three consoles of its generation, Xbox, Playstation 2 and GameCube, the Xbox is the one whose security system has been compromised first, the one that is now easiest to modify for a hobbyist, the one with the most security system workarounds, and the one with the most powerful hacks. This may be, because the Xbox security is the weakest one of the three, but also because Open Source people, homebrew people and crackers attacked the Xbox, while the Open Source people did not attack the Playstation 2, as Linux had been officially supported by Sony, so the total number of hackers was lower, buying them time.

Idea of the Security System

In order to allow only licensed and authentic code to run, it is necessary to build a TCPA/Palladium-like chain of trust, which reaches from system boot to the actual execution of the game. The first link is from the CPU to the code in ROM, which includes the Windows kernel, and the second link is from the kernel to the game.

There are several reasons that the operating system is contained in ROM (256 KB) instead of being stored on hard disk, like on a PC. First, it allows a faster startup, as the kernel can initialize while the hard disk is spinning up, furthermore, there is one link less in the chain of trust, and in case verification of the kernel gets compromised, it is harder to overwrite a ROM chip than modify data on a hard disk.

Startup Security

When turned on, x86-compatible CPUs start at the address 0xFFFFFFF0 in the address space, which is usually flash memory. For the Xbox, this is obviously no good idea, as flash memory can be

• replaced, by removing the chip, fitting a socket and inserting a replacement chip.

• overridden, by adding another flash memory chip to the LPC bus. This override functionality is necessary, because during manufacturing, an empty flash memory chip gets soldered onto the board, an override LPC ROM chip gets connected to the board and the system boots from the external ROM, which then programs the internal flash memory. This procedure is significantly cheaper than preprogramming the flash memory chips.

• reprogrammed, because flash memory can be written to many times. It would be possible to use ROM instead of flash memory, but ROM is more expensive than flash memory.

Thus, the machine must not start from flash memory.

Microsoft's Perspective

It would be possible to make two of the attacks impossible, by using ROM chips instead of flash. There would be no way to reprogram them, and it would be possible to disable the LPC override functionality in the chipset, because it is not needed for the manufacturing process any more.

The Hidden ROM

There is a solution between flash memory and ROM that combines advantages of both these approaches. This trick is rather old and had already been used in previous gaming consoles like the Nintendo 64: Use a tiny non-replaceable startup ROM, and put the bulk of the firmware data (i.e. the Windows kernel) into flash memory. The “internal” ROM checks whether the contents of the flash memory are authentic, and if yes, it passes execution to it.

This way, there will be another link in the chain of trust, but the ROM code can be trusted (if it is non-replaceable), and if, in addition, it is non-accessible, an attacker may not even have a clue how verification works.

Location of the ROM

But where can this ROM be put? It cannot be a separate chip, as it would be replaceable. It would have to be included into another chip. The CPU would be ideal, as the ROM contents would not travel over any visible bus, but then it would be impossible to use cheap off-the-shelf Celerons. Including it in any other chip would make it non-replaceable, but data would travel over a bus. It seems to be a good compromise to store the ROM data in the Southbridge (”MCPX”), as it is connected via the *very* fast HyperTransport bus, so it is very hard to sniff. A former Microsoft employee confirmed that the developers tought that nobody was able to sniff HyperTransport.

Verification Algorithm

This secret ROM stored in the Southbridge must verify the Windows kernel in the external flash memory before executing it. One idea would be to checksum (hash) the flash contents using an algo-

rithm like MD5 or SHA-1, but this would mean that the hash of the kernel has to be stored in the secret ROM as well, which would make it imposible to ship updated versions of the kernel in future Xboxes without also updating the ROM contents - which would be very expensive.

A digital signature algorithm like RSA would be better: It would be possible to update the kernel without changing the ROM, but an RSA implementation takes up a lot of space, and embedded ROM in the Southbridge is expensive. It would be ideal if the algorithm fit in only 512 bytes, which is impossible for RSA.

Second Bootloader (”2bl”)

A solution for this problem is again to introduce another link in the chain of trust: The ROM only hashes a small loader (”2bl”, “second bootloader”) in flash memory, which can never be changed. It is then the job of this loader to verify the rest of flash, and as the second loader can be any size, there are no restrictions.

So the final chain of trust looks like this: The CPU boots from the secret ROM embedded into the Southbridge, which cannot be changed. The secret ROM verifies the second bootloader in flash memory using a hash algorithm, and if it is authentic, runs it. The second bootloader checks the kernel, and if authentic, runs it.

Now the second bootloader and the Windows kernel would be stored in flash memory in plain text, which is a bad idea: An attacker can immediately see how the second bootloader verifies the integrity of the kernel, and even analyze the complex kernel for possible exploits. Encrypting all the flash contents will not solve possible vulnerability problems, but it will buy us time until the decryption of the flash contents is understood by hackers.

The decryption key would have to be stored in the secret ROM, and the 2bl verification code would also have to decrypt the flash contents into RAM while reading it.

RAM Initialization

Decrypting flash memory contents into RAM is a challenge if we are living inside the first few hundred bytes of code after the machine has started up: At this point, RAM might not be stable yet. The reason for this is that Microsoft bought cheap RAM chips; they just took everything Samsung could give them to lower the price, even faulty ones, i.e. chips that will be unstable when clocked at the highest frequencies specified.

The Xbox is supposed to find out the highest clock speed the RAM chips can go and run them at this frequency - this is the reason why some games don't run as smoothly on some Xboxes as on others. So the startup code in the secret ROM has to do a memory test, and if it fails, clock down the RAM, do another memory test, and if it fails again, clock down again, and so on, until the test succeeds or the RAM cannot be clocked down any further.

The problem now is that it is impossible to do complex RAM initialization, data decryption and hashing in 512 bytes. This code would need at least 2 KB, which would be significantly more expensive, if embedded into the Southbridge.

We could put the RAM initialization code, which is the biggest part of what the startup code needs to do, into flash memory, and call it from the secret ROM, but this would kill security, as an attacker could easily see the unencrypted code in flash, modify it and have the control of the machine right at the startup.

The developers at Microsoft had a brilliant idea how to solve this problem: They designed an interpreter for a virtual machine that can read and write memory, access the PCI config space, do “AND” and “OR” calculations, jump conditionally etc. The instruction code has one byte instructions and two 32 bit operands, it can use immediate values as well as an accumulator.

The interpreter for the virtual machine is stored in the secret ROM, and its code (”xcodes”) is stored in flash memory. This code does the memory initialization (plus extra hardware initialization, which would not be necessary). This program cannot be encrypted, as there is again no space for it in the secret ROM, but as the virtual machine is unknown to the hacker, encryption should not be that important. It also cannot be hashed, as this would make it impossible to change the xcodes for later revisions of the Xbox hardware. Therefore we have to make sure that, if the hacker knows how the virtual machine works, it is impossible to do anything malicious with the xcodes.

The Virtual Machine

There are several ways an attacker could exploit the xcodes, which are by definition untrusted, because they reside in “external” flash memory. Microsoft included some code to make sure there were no possible exploits.

Read the Secret ROM

The xcodes can read memory and access I/O ports. This way an attacker could place xcodes into flash memory that dump the secret ROM, which must be mapped into the address space somewhere, to a slow bus, like the LPC or the I2C bus, or write it into CMOS or the EEPROM, so that we can read it later.

The xcode interpreter has to make sure that the xcodes cannot read the secret ROM, which is located at the upper 512 bytes of the address space. The simplest way to accomplish this is to mask the address when reading from memory:

```
  and ebx, 0FFFFFFFh; clear upper 4 bits
  mov edi, [ebx]    ; read from memory
  jmp next_instruction
```

This way, the xcodes can only ready from the lower 256 MB, which is no problem, as there are only 64 MB of RAM, and memory mapped I/O can be mapped into this region using PCI config cycles.

Turn off the Secret ROM

The xcodes may also not turn off the secret ROM, or else the CPU, while executing the xcode interpreter, would "fall down" from the secret ROM into the underlying flash ROM, which is also mapped to the top end of the address space. The turn off functionality is important: As soon as the second bootloader takes over, the secret ROM has to be turned off, or else an attack against a game, which makes it possible to run arbitrary code, could dump the secret ROM, making additional attacks against it possible.

The secret ROM can be turned off by writing a value with bit #1 set to the PCI config space of device 0:1:0, register 0x80. So the xcode interpreter always clears this bit in case there is a write to this PCI config space register:

```
  cmp ebx, 80000880h       ; MCPX disable?
  jnz short not_mcpx_disable; no
  and ecx, not 2           ; clear  bit 1
not_mcpx_disable:
  mov eax, ebx
  mov dx, 0CF8h
  out dx, eax  ; PCI configuration address
  add dl, 4
  mov eax, ecx
  out dx, eax  ; PCI configuration data
  jmp short next_instruction
```

Encryption and Hashing

For the decryption of the second bootloader, Microsoft chose the RC4 algorithm, which is pretty small, as it fits into 150 bytes. It uses a 16 bytes key, which is also stored in the secret ROM. Microsoft's engineers also chose to use RC4 as a hash, so that no additional algorithm had to be implemented for this. Differential decryption algorithms feed the decrypted data into the generator of the decryption key stream, so if the encrypted code is changed at one byte, all the following bytes will decrypted incorrectly, up to the last bytes. This way, it is possible to only test the last few bytes. If they have been decrypted correctly, then the encrypted code has been authentic. (If you are getting suspicious now - read on!)

In practice, the secret ROM in the Xbox compares the last decrypted 32 bit value with the constant of 0x7854794A. If it is incorrect, the Xbox has to panic.

Panic Code

So far, the code in the secret ROM does this:

- Enter protected mode, and set up segment descriptors, so that we have access to the complete flat 32 bit address space.
- Interpret the xcodes.
- Decrypt and hash the second bootloader, store it in RAM
- If the hash is correct, jump to the decrypted second bootloader in RAM, else panic.

There is another possible attack here: A hacker could deliberately make the hash fail. If the Xbox then halts and flashes its lights to indicate an error, the attacker can attach a device to dump the secret ROM after the CPU has shut down and the bus is idle. Although HyperTransport is fast, it would be a lot easier to attach a device that actively requests the data from the Southbridge than sniffing it when the CPU requests it.

One solution would be not to halt but to shut down the Xbox in case of a problem. The support chips have this functionality. But incorrect flash memory does not necessarily mean that there has been an attack, it could also be a malfunction, and the machine should use the LED to blink an error code.

So we should leave the Xbox running, but just turn off the secret ROM, so that it cannot be read any more. But there is a problem: We have to do this inside the secret ROM. So if we disable the ROM, we cannot have the "hlt" instruction after that, because the CPU will "fall down" into flash memory - where an attacker could put code. On the other hand, if we halt the CPU, we cannot turn off the secret ROM afterwards.

We cannot put the disable and halt code into RAM and jump there, because RAM might not be stable, and might even have been tampered with by an attacker (e.g. by turning off the memory controller using the xcodes) so that the secret ROM does not get turned off. We cannot put the disable and halt code into flash either, as again, an attacker could simply put arbitrary code to circumvent the complete system there.

The Microsoft engineers used yet another brilliant trick: They jump to the very end of the address space (which is covered by the secret ROM) and turn off the secret ROM in the very last instruction

inside the address space. This is a simplified version of the idea:

```
FFFFFFF1    mov eax, 80000880h
FFFFFFF6    mov dx, 0CF8h
FFFFFFF9    out dx, eax
FFFFFFFB    add dl, 4
FFFFFFFC    mov al, 2
FFFFFFFE    out dx, al
```

After the last instruction, the program counter (EIP) will overflow to 00000000, which, according to the CPU documentation, causes an exception, and as there is no exception handler set up, it causes a double fault, which will effectively halt the machine.

The Hacker Perspective

So much for the theory. The design looked pretty good, although the trade off between cost and security as it has been decided, might give some people headaches. Let us now have a look at the Xbox from the hackers' point of view.

It has been well known that the Xbox chipset is a modified version of nVidia's nForce chipset, so we knew that it was standard IDE, USB, there was an internal PCI bus and so on. Two hackers from Great Britain, Luke and Andy, checked the hard disk and found out that it uses a custom partitioning scheme, a FAT-like filesystem, that there is no kernel on the hard disk, but there is the Xbox Dashboard on the fourth partition, the main program that gets executed if there is no game in the DVD drive, which allows changing settings, playing audio CDs and managing savegames.

Extracting the Secret ROM

Andrew “bunnie” Huang, then a PhD student at the MIT, disassembled his Xbox, saw the flash memory, de-soldered it, extracted the contents, put it on his website and got a phone call from one of Microsoft's lawyers.

The flash memory image was obviously encrypted, but there was x86 binary code in the upper 512 bytes! Obviously, there should be no code in the upper 512 bytes, as this gets overridden by the secret ROM, which contains the actual machine setup and flash decryption code.

Bunnie found out that this code was an interpreter for tables in flash memory, plus a decryption function that looked like RC4. He rewrote the crypto code in C and tried it on the data - but the resulting data was random, obviously something was wrong. The interpreter didn't make much sense either. The code used opcodes that were unknown to the interpreter.

In order to find out what was wrong, bunnie rewrote the top of flash with his own code, and later even completely erased the upper 512 bytes, but the Xbox still booted! So it was obvious to him that this region gets overridden by some internal code. As it turned out later, the code in the upper 512 bytes of the flash image was a very old version of the secret ROM code, which had been unintentionally linked to the image by the build tools. It seems like nobody had looked at the resulting image at the end, before they shipped the consoles. This mistake was very close to a fatal one, and Microsoft was lucky that they didn't link the actual version of the secret ROM.

But it didn't make that much of a difference, as bunnie sniffed the busses, and eventually dumped the complete secret ROM, including the RC4 key from HyperTransport, using a custom built sniffer - after all, he was working on his PhD degree about high performance computing, and he could use the excellent resources of the MIT hardware lab.

When he published his findings, other people found out quite quickly that the validity check did nothing at all: The combination of decryption and hash with a cypher that feeds back the decrypted data into the key stream is a good idea, but unfortunately, RC4 is no such cypher. It decrypts bytes independently, so if one byte is wrong, all the following bytes will still be decrypted correctly. So checking the last four bytes has no effect: There is no hash.

It turned out that the cypher used in the old version of the secret ROM as found in flash memory used the RC5 cypher. In contrast to RC4, RC5 does feed the decrypted stream back into the key stream. So they seem to have replaced RC5 with RC4 without understanding that RC4 cannot be used as a hash. Bunnie's theory why they abandoned RC5 is that RC5 was still a work in progress, and that Microsoft wasn't supposed to have it, so they went for the closest relative - RC4.

Modchips

Now that the encryption key was known and there was effectively no hash over the second bootloader, it was possible to patch this code: People added code to the second bootloader to patch the kernel after decryption (and decompression) to accept executables even if on the wrong media (DVD-R instead of original) or if the RSA signature of the executables was broken (i.e. unsigned homebrew software).

Modchips appeared: Some of them had a complete replacement flash memory chip on them, others only patches a few bytes and passed most reads down to the original flash chip. All these modchips had to be

soldered in parallel to the original flash chip, using 31 wires.

Now other people found out that, if the flash chip is completely missing, the Xbox wants to read from a (non-existant) ROM chip connected to the (serial) LPC bus. This is of course because of the manufacturing process: As it has been explained before, the flash chip gets programmed in-system, the first time they are turned on, using an external LPC ROM chip. Modchip makers soon developed chips that only needed 9 wires and connected to the LPC bus. It was enough to ground the data line D0 to make the Xbox think that flash memory is empty.

Lots of these “cheapermods” appeared, as they only consisted of a single serial flash memory chip. They could be installed within minutes, especially after some companies started shipping chips that used pogo pins, so that no soldering was required.

Some groups wrote applications like boot menus that made it possible to copy games to hard disk and run them from there. Patched Xbox kernels appeared that supported bigger hard disks. Making the Xbox run copies from DVD-R or hard disk as well as homebrew applications written with the official Xbox SDK was now easy.

Backdoors

The Xbox Linux Project was working on two ways to start Linux: Either run the Linux kernel from a CD/DVD as if it was a game, or run it directly from flash memory, or from HD/DVD using a Linux bootloader in flash memory, so that the Xbox behaved like a PC. For the latter, Xbox Linux was working on a replacement firmware.

It would have been no problem to write a replacement firmware that took over execution instead of the second bootloader, as it was possible to completely replace this second bootloader, as well as encrypt it, using the well-known key from the secret ROM. But the firmware developers felt very uncomfortable with the idea of using this secret key in their GPL code. Other hackers felt the same, and thus were looking for bugs and backdoors in the secret ROM code, in order to find a way to be able to implement a replacement firmware without having to deal with encryption.

The Visor Backdoor

A hacker named visor, who never revealed his real name, wondered whether the rollover to 00000000 in case of an incorrect 2bl “hash” really caused a double fault and halted the CPU. He used the xcodes to write the assembly instruction for “jmp 0xFFFF0000” to the memory location 00000000 in RAM and changed the last four bytes in 2bl, in order to make the secret ROM run the panic code. The Xbox happily continued executing code at 00000000 and took the jump into flash.

When appending these instructions to the existing xcodes, he could make sure that RAM had been properly initialized and was thus stable. So there was no need to encrypt the Xbox Linux bootloader firmware with the secret key any more. It was enough to add the memory write instruction to the end of the xcodes and make sure that 2bl decryption fails - which will automatically happen, if the firmware replacement does not contain the 2bl code.

Now why is there no double fault? Hackers from the Xbox Linux team checked with AMD employees and they explained that AMD CPUs *do* throw an exception in case of EIP overflows, but Intel CPUs don't.

The reason that Intel CPUs don't is because of... 1970s stuff. Execution on x86 CPUs starts at the top of the address space (minus 16 bytes), but some computer makers wanted to have their ROM at the bottom of the address space, i.e. at 0, so Intel implemented the instruction with the encoding 0xFFFF, which is what you get when reading from addresses not connected to any chip, as a No-Operation (”nop”) and made the CPU throw no exception in case of the address space wraparound. This way, the CPU would “nop” its way up to the top, and finally execute the code at 0.

AMD did not implement this behavior, as it had not been necessary any more by the time AMD entered the x86 market with it own designs, and because they felt that this behavior was a security risk and fixing it would not mean a significant incompatibility.

But why did Microsoft do it wrong? This can be explained with the history of the Xbox: AMD offered to design and manufacture both the CPU and the motherboard (including the chipset), and nVidia was contracted to contribute the graphics hardware. The first developer systems, even outside of Microsoft, were Athlon-based, but then Intel came in and offered their chips for less money, as well as the complementary redesign of the existing AMD chipset to work with their CPU. Consequently, nVidia licensed the AMD chipset so that the AMD name vanished. This also means, that nVidia nForce chipset is essentially AMD technology, closely related to the AMD-760 chipset.

So when Microsoft switched from AMD to Intel, they apparently forgot to test their security code again with the new hardware, or to read the Intel datasheets.

The MIST Hack

Soon after the visor hack, another vulnerability was found in the secret ROM code, attacking the

code that checks whether an xcode wants to disable the secret ROM. Let us look at this code again:

```
 cmp ebx, 80000880h       ; MCPX disable?
  jnz short not_mcpx_disable; no
  and ecx, not 2           ; clear  bit 1
not_mcpx_disable:
  mov eax, ebx
  mov dx, 0CF8h
  out dx, eax  ; PCI configuration address
  add dl, 4
  mov eax, ecx
  out dx, eax  ; PCI configuration data
  jmp short next_instruction
```

The PCI config address is stored in the EBX register in the beginning. This address has to be sent to I/O port 0x0CF8, and the 32 bit data has to be sent to I/O port 0x0CFC. The address is encoded like this:

```
0-7    reg
8-10   func
11-15  device
16-23  bus
24-30  reserved
31     always 1
```

The attack is pretty obvoius: there are seven reserved bits in the address, and the code tests for a single exact value. What happens if we write to an alias of the same address, by using an address with only some of the bits 24 to 30 changed? While the instruction

```
POKEPCI(80000880h, 2)
```

will be caught, the instruction

```
POKEPCI(C0000880h, 2)
```

will not be caught - and works just as well, because the PCI bus controller just ignores the unused bits.

This instruction disables the secret ROM, that is, the interpreter disables itself when sending the value to port 0x0CFC, and the CPU falls down to flash memory. We can put a “landing zone” into flash, by filling all of the top 512 bytes with “nop” instructions, and putting a jump to the beginning of flash into the last instruction, so that we do not have to care where exactly the CPU lands after falling down, and we are independent of possibly hard to reproduce caching effects.

It is hard to find a good reason for this bug other than carelessness. It might be attributed to not reading the documentation closely enough, as well as not looking at it from the perspective of a hacker well enough. After all, this code had been written with a specific attack in mind - but the code made hacking easier, by giving hackers a hint how to attack.

Another PCI Config Space Attack

There is a second sequence of xcode instructions that can disable the secret ROM just as well, which are not caught by the interpreter: The interpreter supports writing bytes to I/O ports, so it is possible to put together the code to disable the secret ROM using 8 bit I/O writes:

```
OUTB(0xcf8), 0x80
OUTB(0xcf9), 0x08
OUTB(0xcfa), 0x00
OUTB(0xcfb), 0x80
OUTB(0xcfc), 0x02
```

This hack has been unreleased until now. It has been found not long after the MIST hack, but kept secret, in case Microsoft fixed the MIST bug. In the meantime, they have implemented a fix that makes all hacks impossible that are based on turning off the secret ROM. This will be described in detail later.

More Ideas

There have been more ideas, but few of them have been pursued, as long as other existing backdoor existed. One possible idea is to base a hack on caching...

Startup Security, Take Two

When bunnie hacked the secret ROM, Microsoft reacted by updating the ROM. Thousands of already manufactured Southbridges were trashed, new ones made. The hacker community called these Xboxes “version 1.1” machines.

Microsoft's Perspective

Microsoft had now understood that RC4 cannot be used as a hash, so they implemented an additional hash algorithm, which was to be executed after decryption. As there were only few bytes left, the hash algorithm had to be tiny - so the “Tiny Encryption Algorithm” (”TEA”) was used. Every encryption algorithm can be changed to be used as a hash, and TEA seemed to be a good choice, as it is really small. While they were at it, they also changed the RC4 key in the secret ROM, so that hackers would not be able to decrypt 2bl and the kernel without dumping the new secret ROM.

The Hacker Perspective

The extraction of the secret ROM was done by members of the Xbox Linux Project this time, only days after they got their hands on the new 1.1 boxes, and only two weeks after they first appeared.

The A20 Hack

To date, Microsoft does not know how the Xbox Linux Project did it. But since there will most probably be no future revisions of the Xbox, as the Xbox 360 has already taken over, we can release this now.

Let us start with some PC history. The 8086/8088, the first CPU in the x86 line, was supposed to be as closely compatible to the 8080, which was very successful on the CP/M market. The memory model therefore was similar to the 8080, which could access only 64 KB, by dividing memory into 64 KB blocks. Intel decided that the 8086/8088 could have a maximum of 1 MB of RAM, which would have meant 16 “segments” of 64 KB each. But instead of doing it this way, they decided to let the 64 KB segments overlap, and have 65536 of these segments, starting every 16 bytes.

An address was therefore specified by a segment and an offset. The segment would be multiplied by 16, and the offset would be added, to result in the effective address. As an example, 0x0040:0x006C would be 0x40*0x10+0x6C=0x46C. An interesting side effect of this method is that it is possible to have addresses above 1 MB: The segment 0xFFFF starts at the effective address 0xFFFF0, so it should only contain 16 bytes instead of 64 KB. So the address 0xFFFF:0x0010 would be at 1 MB, and 0xFFFF:0xFFFF would be at 1 MB plus roughly 64 KB.

The 8086/8088 could not address more than 1 MB, because it only had 20 address lines, so addresses above 0xFFFF:0x000F were wrapped around to the lower 64 KB. But this behavior was different on the 286, which had 24 address lines: It was actually possible to access roughly 64 KB more using this trick, which was later abused by MS-DOS as “high memory”.

Unfortunately there were some 8086/8088 application that broke, because they required the wrap-around for some reason. It wasn't Intel who found that out, but IBM, when they designed the IBM AT, and it was too late to modify the behavior of the 286, so they fixed it themselves, by introducing the A20 Gate (”A20#”). An unused I/O pin in the keyboard controller was attached to the 20th address line, so that software could pull down address line 20 to 0, thus emulating the 8086/8088 behaviour.

This feature was later moved into the CPUs, and all Pentiums and Athlons have it - and so does the Xbox. If A20# is triggered, bit 20 of all addresses will be 0. So, for example, an address of 1 MB will be 0 MB, and if the CPU wants to access the top of RAM, it will actually access memory that is 1 MB lower than the top.

Keeping this in mind, the attack on the Xbox is pretty straightforward: If we connect the CPU's A20# pin to GND, the Xbox will not start from FFFFFFF0, but from FFEFFFF0 - this is not covered by the secret ROM, but is ordinary flash memory, because flash is mirrored over the upper 16 MB. So by only connecting a single pin, the secret ROM is completely bypassed.

What is cool about this, is that the secret ROM is still turned on. So we could easily dump the secret ROM trough one of the low speed busses (we used the I2C bus), by placing a small dump application into flash memory.

The TEA Hash

After reading Bruce Schneier's book on crypto, we learned that TEA was a really bad choice as a hash. The book says that TEA must never be used as a hash, because it is insecure if used this way. If you flip both bit 16 and 31 of a 32 bit word, the hash will be the same. We could easily patch a jump in the second bootloader so that it would not be recognized. This modified jump lead us directly into flash memory.

But why did they make this mistake? Obviously the designers knew nothing about crypto - again! - and just added code without understanding it and without even reading the most basic books on the topic. A possible explanation why they chose TEA would be that they might have searched the internet for a “tiny” encryption algorithm - and got TEA.

Visor Backdoor and MIST Hack

The Visor Backdoor was still present, so again, for the replacement Linux firmware, the Xbox Linux developers did not have to exploit the crypto code, but could simply use this backdoor. Microsoft obviously released the updated secret ROM much too quickly, just after bunnie dumped it and people saw that RC4 was no hash, but before the visor backdoor had been discovered.

The MIST hack had been discovered after the visor bug - but it no longer worked on the Xbox 1.1. Not because they fixed the comparison - they didn't -, but because they changed the address logic: If you accessed the upper 512 bytes of the address space, and the secret ROM was turned off, the Xbox would just crash, thus making all “fall down” hacks impossible. This way they closed both possible attacks, writing to an alias, and using 5 OUTB instructions.

Microsoft obviously discovered the turnoff vulnerability themselves, closing at least one backdoor, but keeping another one open, and not really closing a second one. It was too expensive to trash the 1.1 Southbridge chips again for yet another update, so Microsoft still uses these chips in today's Xboxes.

Today

In later revisions of the Xbox, Microsoft removed some pins of the LPC bus, making modchip design harder, but they could not remove the LPC bus altogether, because they needed it during the manufacturing process.

In the latest revision of the Xbox hardware (v1.6), they finally switched from flash memory to real ROM - and even integrated the ROM with the video encoder. The LPC bus is not needed for manufacturing any more, as the ROM chips are already preprogrammed. So now it is impossible to replace or to overwrite the kernel image, and because of the missing LPC bus, it also seems impossible to attach a ROM override.

But modchips are still possible. The obvious LPC pins are gone now, but the bus is still there. If you find the LPC pins on the board, you can attach a ROM override just as before, the modchips are only a bit harder to install. This is because the Southbridge still has the LPC override functionality, since they did not make a new revision of it - as so often, obviously for monetary reasons.

Xbox Kernel Security

Let us have a look at the chain of trust again:

- The CPU starts execution of code stored in the secret ROM.
- The secret ROM decrypts and verifies the second bootloader.
- The second bootloader decrypts and verifies the Windows kernel.
- The Windows kernel checks the allowed media bits and the RSA signature of the game.

This last link is a complete software thing, so all the attacks have been pretty much standard. Some people tried to brute force the RSA key used for the game signature - no joke! But what is more likely, successfully brute forcing RSA 2048, or finding a bug in Microsoft's security code? After the experience with the first links of the chain of trust, the Xbox Linux Project focused on finding bugs in the software.

We found no bug in the RSA implementation. It is taken straight out of Windows 2000 and looks pretty good. But there are always implicit additional links in the chain of trust: All code reads data, and data can cause security risks if handled incorrectly.

Game Exploits

What data do games load? Graphics data, audio data, video data... - but we cannot alter them, because it is not easily possible to create authentic Xbox DVDs, and the Xbox won't boot originals from DVD-R etc.

But most games can load savegames, and these can easily be changed: The Xbox memory units are more or less standard USB storage devices (”USB sticks”), so it is possible to use most USB sticks with the Xbox, and just store hacked savegames on them.

Plenty of Xbox games had buffer vulnerabilities in their savegame handlers. It was often as easy as extending the length of strings like the name of the player, and the game would overwrite its stack with our data and eventually jump to the code we embedded in the savegame.

The procedure for the user was then to simply copy a hacked savegame from a USB stick onto the Xbox hard disk, run the game and load the savegame. But after a buffer exploit, we would normally only be in user mode - not on the Xbox, as all Xbox games run in kernel mode. The reason for this is probably a slight speed advantage, or, less likely, a simpler environment for the game, but Microsoft tried to make the environment as similar to the Windows/DirectX environment as possible, so user mode would have actually made the environment “simpler” for many Windows/DirectX developers.

Now that we have full control of the machine, we can overwrite the flash memory chip. It is write protected by default, but disabling the write protection is as easy as soldering a single bridge on the motherboard. After all, this bridge has to be closed temporarily during manufacturing when programming flash memory for the first time. Using this hack, it is possible, only with a USB stick, one of several games (007 Agent Under Fire, MechAssault, Splinter Cell, ...) and a soldering iron, to permanently modify the Xbox, just as if a modchip was installed. Because early Xboxes had a 1 MB flash chip, although only 256 KB had been used, it was even possible to install several ROM images in flash and attach a switch.

But the Xbox Linux Project did not blindly release this hack. The first savegame proof of concept exploit had been finished in January 2003. After that, a lot of energy was invested in finding out a way to free the Xbox for homebrew development and Linux, but not allowing game copies. Microsoft was contacted, but without any success. They just ignored the problem.

Finally in July, the hack was released, with heavy obfuscation, and lockout code for non-Linux use. It was obvious that this would only slow down the “hacking of the hack”, so eventually, people would be able to use this vulnerability for copied games, but since Microsoft showed no interest in finding a solution, there was no other option than full disclo-

sure. The suggestion of the Xbox Linux Project would have been to work together with Microsoft to silently close the security holes and, in return, work on a method to let homebrew and Linux run on the Xbox.

Dashboard Exploits

The problem with the savegame hack was that, if you didn't want to overwrite the flash memory chip, you had to insert the game and load the savegame every time you wanted to run unsigned code. But having full control of the machine using the savegame exploit also meant we could access the hard disk without opening the Xbox. This way, it became interesting to closely examine the hard disk contents for vulnerabilities.

The Dashboard is the main program on hard disk, executed every time the Xbox is started without a game in the DVD drive. The dashboard may even be the very reason the Xbox ships with a hard disk: While the settings menu and savegame management on the Nintendo GameCube fit well into 2 MB of ROM, the Xbox Dashboard, which is roughly comparable in its functionality, occupies more than 100 MB. So the original idea why to include a hard disk might have been initiated by the inability to compress the Dashboard into typical ROM sizes - and they might have decided to make the best out of it, and find additional uses for the hard disk.

The dashboard loads its data files, like audio and graphics, from hard disk. With the savegame exploit, we can now alter the hard disk contents, even without opening the Xbox. Of course the dashboard executable is signed and can therefore not be altered, and all data files are hashed, with the hashes stored inside the dashboard executable. Well, all files, except for two: the font files.

Consequently, there was an integer vulnerability in the font handling routines, so that we could run our own code by replacing the font files. Combined with the savegame exploit, it was as easy as transferring the savegame and loading it, which would run a script that modifies the fonts.

Now every time the Xbox is turned on, the Dashboard crashes because of the faulty fonts and runs our code embedded in these files. Our code reloads the Dashboard with the original fonts, hacks it, and runs it. Hacking the Dashboard meant two things: Modifying one menu entry to read “XBOX LINUX” instead of “XBOX LIVE” and running the Linux bootloader instead of the Xbox Live setup executable, and modifying the kernel to accept both applications signed with Microsoft's RSA key as well as those signed with our RSA key, from hard disk and from CD/DVD. We called this “MechInstaller”, as it was based on the “MechAssault” savegame exploit.

Only accepting code either signed by the original key or by our key, keeping our key secret, and using heavy obfuscation again, meant that nobody could easily abuse this solution for copied games.

This hack shows several things: Hackers have phantasy, the combination of flaws can lead to fully compromising the security system, powerful privileged code should be bug-free and security code should really catch *all* cases.

Oh, and there is another vulnerability, and integer vulnerability in the audio player code. The attack was developed independently of the font attack, but was inferior because it would have required the user to enter the audio player every time to run Linux.

Microsoft's Fixes

The history of Microsoft's reactions to the font vulnerability is the perfect lesson of how to do it wrong.

1. After MechInstaller had been released, Microsoft fixed the buffer vulnerability in the Dashboard and distributed this new version over the Xbox Live network and shipped it with new Xboxes.
2. For the hackers, this was no major problem: It was possible to downgrade the Dashboard of a new Xbox to the vulnerable version. Just run Linux using a savegame exploit, and “dd” the old image. Some people felt downgrading on new Xboxes was not piracy, because after all, Microsoft upgraded Xbox Live users' hard disks to the new version without asking.
3. As the next step, Microsoft blacklisted the old Dashboard in the new kernel. It was impossible to just “dd” an old Dashboard image onto newer Xboxes.
4. Still no major problem for hackers: The second executable on the hard disk, “xonlinedash”, which is used for Xbox Live configuration, had the same bug, so it was possible to copy the old “xonlinedash” and to rename it to “xboxdash” to make it crash because of the faulty fonts.
5. Microsoft consequently blacklisted the vulnerable version of “xonlinedash”.
6. Again, no major problem for hackers: All Xbox Live games come with the “dashupdate” application, which adds Xbox Live functionality to the Dashboard for the first Xboxes which came without it. This update application has the same font bug, and it can be run from hard disk. So it is possible to copy the file from any Xbox Live game DVD, rename it to “xboxdash” and let it crash.

7. Microsoft could not blacklist this one. Xbox Live enabled games run the update application every time they start, making sure the Xbox has the Xbox Live functionality. Blacklisting "dashupdate" would break these games.

We won.

The Mistakes that Have Been Made

Microsoft obviously made a lot of mistakes. But it would be too easy to just attribute all these to stupid engineers. There have been good (and different) reasons for most of these mistakes, and one can learn a lot from them.

There are 17 kinds of mistakes they made, several of which have been made more than once. I will group the 17 mistake types into three categories: Design mistakes, implementation mistakes and pad policy decisions.

Design

#1: Security vs. Money

Be very careful with tradeoffs between security and money. There are rarely sensible compromises. Keep in mind that the very reason for the security system is to make more money, or to prevent money losses. Security systems cannot be "a little better" or "a little worse". Either they are effective - or they are not. By saving money on the security system, you may easily make it not effective at all, not only wasting the money spent on the security system, but also making losses because it is not effective.

Microsoft made many compromises.

- In-system programming of flash memory is cheaper than preprogramming, but an attacker can also override the firmware with an LPC ROM.
- Buying all of Samsung's RAM chips is cheaper than only buying those within the specs, but it made RAM initialization more complex, using up space that could otherwise be used for better security code.
- They chose to put the secret ROM into the Southbridge instead of the CPU, because the Southbridge was a custom component anyway and having a custom CPU would have been a lot more expensive, but keys travel over a visible bus if the secret ROM is outside the CPU.
- They saved money choosing not to update the Southbridge a second time, which would have fixed the TEA hash and removed the visor backdoor. This would have made modchips virtually impossible.

#2: Security vs. Speed

Don't trade security for speed. Although it may be true that the product in question must be as fast as possible in order to be able to compete with similar products on the market, remember that in IT, computers aren't slower or faster by some percentage - but but factors! Besides, you might lose more money because of a security system that does not work than because of a product that is 10 percent slower than it could be.

Most probably for added speed (one address space, no TLB misses), Microsoft chose to run all code in kernel mode, even games that interacted with untrusted data that came from the outside. This made it possible to have complete control of the machine once a game crashed because of a prepared savegame, including complete control of the hard disk and the possibility of booting another operating system.

#3: Combinations of Weaknesses

Be aware of the fact that a combination of security flaws can lead to a successful attack. Don't think that a possible security hole (or "only" a security risk) cannot be exploited because there are so many barriers in front of it. Attackers might break all the other barriers that block the vulnerability, and fixing that one hole would have stopped them.

MechInstaller is a great example for that. It was only possible because of the combination of several security weaknesses:

- The boot process was vulnerable, so we could use a modified kernel to analyze games.
- Some games are not careful enough with savegames, so that we can run our own code.
- Games run in kernel mode, so we have full control of the hardware.
- The Dashboard does not verify the integrity of the font files.
- The Dashboard has a vulnerability in the font code.

If any of these weaknesses had not been there, then MechInstaller would not have been possible. Also note that hackers have enough fantasy to find out these combinations.

#4: Hackers' Resources

Understand that hackers may have excellent resources. Hobbyists may use resources from work or from university, and professional attackers can also be very well-equipped. It is a big mistake to underestimate them. So never think you are safe because it would be too much work or too expensive to exploit a weakness. If it is a weakness, it will eventually be exploited. Also understand that hackers may

have excellent human resources. Not only in number, but also in qualifications.

Microsoft put the secret ROM into the Southbridge instead of the CPU, which meant that the secret key would travel over a visible bus. This is the very fast HyperTransport bus, which, at that time, could not be sniffed using logic analyzers any mortal could afford. But with help of the resources of the MIT and using all of his expertise, bunnie could build his own hardware that could sniff the bus.

#5: Barriers and Obstacles

Don't make anything “harder for hackers”. Instead make it “impossible for hackers”, or, if it cannot be made impossible, don't care about it. Because of the potential great number and excellent qualifications of hackers, no obstacle will have any effect or slow down hacking significantly. But instead, in security design, you might make mistake #3, because you think you are safe as there are so many obstacles in the hackers' way. Use the resources you would invest into building obstacles into building or strengthening barriers instead - possibly at a different location.

Microsoft built obstacles into the system at many different locations.

- Savegames will only be accepted if they are signed, but the private key is of course stored inside the game, so this is no barrier. Instead, they should have made sure the games contain no buffer vulnerabilities in their savegame handlers.
- The hard disk is secured with an ATA password, different for every Xbox and stored on an EEPROM inside the Xbox, but an attacker can just “hotswap” an unlocked hard disk from a running Xbox to a running PC. Instead, they should have put that energy into verifying whether the Dashboard really hashes all data it reads from the hard disk.
- The 512 bytes of security startup code were embedded in a custom chip to make it hard to sniff. Instead, they should have made sure that there are no bugs in that security code.

#6: Hacker Groups

Don't use one security system for different purposes, or else attackers with very different goals will jointly attack it, being a lot more effective. Instead, try to find out who your enemies really are and what they want, and design your security system so that every group gets as much of what they want so that it does not hurt you.

There were three possible goals for Xbox hackers: Run Linux and use it as a computer, run homebrew software like media players and emulators, and run copies Although there were some overlaps between Linux and homebrew people, as well as between homebrew people and people interested in copies, these were essentially three very different groups. Because they were all locked out by the same protection, they worked together, either explicitly, or implicitly, by using the results of each other. No Linux hackers ever attacked the Playstation. When you are fair, people don't fight you.

#7: Security by Obscurity

Security by obscurity does not work. Well-proven algorithms like SHA-1 and RSA work (of course given your implementation is well-proven as well).

Microsoft hid the secret ROM, the Windows kernel, the game DVD contents (no way to read them on a standard DVD drive) and the hard disk contents using different methods. None had any effect. Also see #5.

#8: Fixes

When your security system has been broken, don't release quick fixes, for two reasons: Your fixes may be flawed and may not actually correct the problem, and even worse holes may be found not much later, which you must fix again - and ship yet another version. Instead, every time a security vulnerability is found, audit your complete security system and search for similar bugs, as well as other bugs in the same part of the system, based on the knowledge you gained from the successful hack.

Microsoft failed to correct the hash problem in the second version of the secret ROM, and didn't fix the visor vulnerability, which was found just weeks later. After trashing thousands of already manufactured v1.0 Southbridge chips, which was very expensive, they decided not to update the Southbridge a second time. Another example is the dashboard odyssey: Instead of blacklisting the vulnerable executables at a time, they released three updates, none of which was effective.

Implementation

#9: Data Sheets

Know everything about the components you use. Do read data sheets. Be very careful with components that have legacy functionality.

Microsoft did not notice the A20# legacy functionality as a security risk. It seems that they did not completely analyze the functionality of the Pentium III Celeron, or else they should have noticed. They also apparently did not read the Pentium programmers' manual, or else they would have noticed that Intel CPUs do not panic on a FFFFFFFF/00000000 wraparound.

#10: Literature

Read (at least!) standard literature. If you are dealing with cryptography, this means you have to read at last Schneier's "Applied Cryptography".

Microsoft's engineers did not know that TEA must not be used as a hash, and that RC4 does not feed the decrypted stream back into the key stream.

#11: Pros

Get experienced professionals to work on your security system, both on the design and the implementation. If it's a money issue, see #1.

Looking at mistakes #9 and #10, it seems very probable that at least some of Microsoft's engineers had no prior experience with cryptography or the design of a security system. We also know that people on an internship were working on Xbox security.

#12: Completeness

Check whether your security code catches all cases. If it does not, you did not only waste time implementing all of it, but you may also give hints to hackers: If there are many checks at one point of the code, it looks a lot like code that is relevant for security and an attacker can check whether all cases are caught.

Microsoft made this mistake twice: The xcode interpreter tests for the secret ROM turnoff code, and doesn't catch all cases. And the Dashboard hashes all files it is going to read, except for two. This gave us the ideas where to attack.

#13: Leftovers

Look at the final product from the perspective of a hacker. Hexdump and disassemble your final builds. There could be leftovers!

The Xbox flash memory image contained an old version of the secret ROM, giving us not only hints about the contents of the actual secret ROM, but also an insight into what Microsoft planned and why some mistakes have been made.

#14: Final Test

Test your security system when you have the final parts and with the final software components in place. Changing something may very well open holes somewhere else. When you change something, rethink the complete system, and check all assumptions that you made.

The visor hack was only possible because Microsoft failed to adapt their security system, designed for the AMD CPU, to the Intel CPU. The "hash" in the secret ROM had no effect because they changed RC5 to RC4 without thinking about the implications.

Policies

#15: Source

Keep your source safe. Find engineers you can trust.

The complete Xbox source code has leaked, including the kernel and libraries source. Groups interested in copies could easily modify it to support running games from hard disk, support for hard disks bigger than 137 GB, custom boot logos etc. This had been previously done by patching the binary.

#16: Many People

Have many good people have a look at both your design and your implementation. Keeping your source code safe means having engineers you can trust, and not letting none of your engineers see the source code. As stated at #7, your system should not rely on the source code being safe. Unless you did #7 completely wrong, a bug in the security system is typically a lot worse than a leak of the source code.

It seems a lot like very few people have actually seen the Xbox security code.

#17: Talk

Know your enemy - and talk to them. They are not terrorists that you are not supposed to negotiate with. Their intent is not to harm you but to reach their goals. Working on their goals on their own might harm you indirectly, because the hackers may not care about the same things as you do. Seek the contact to hackers, know what they are doing and have them inform you about a vulnerability before publishing it. Make them know your position and why they should respect it, but also respect their position. Offer them to loosen the security system for what they want in exchange for the non-disclosure of their findings.

Microsoft refused to talk about the savegame and font vulnerabilities. If we had been bad hackers, we could have released both of them as-is, immediately making it possible to run copies on Xboxes without the use of a modchip. Instead, we sought contact to Microsoft: We would have preferred to see a backdoor for Linux in the Xbox security system, instead of a solution based on our findings that would allow running copies. But as they refused to talk, we were forced to release the exploits, and they were lucky we heavily obfuscated our solutions so in order to slow down people interested in using it for copies.

Conclusion

The security system of the Xbox has been a complete failure.