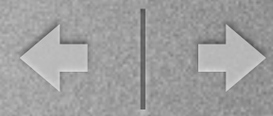
Trusted Computing & the XBox

Aggelos Kiayias

XBox

- A full featured PC:
  - Pentium III celeron, 733 MHz
  - 64MB RAM
  - GeForce 3 with TV out
  - 10 GB IDE Hard Disk/ IDE dvd
  - Fast Ethernet / USB
- Simplified Windows 2000 Kernel/ Static Win32, libc, DirectX

Motivation for Security

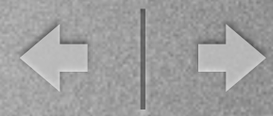
- Prevent sales as a general purpose PC.
- Prevent development of unauthorized games or media applications.
- Prevent duplication of games.
  - the above are dictated by the business model of distributing XBox


Trusted Computing

- Employ trusted computing techniques to make sure that the general purpose hardware of an XBox can only be used in the specialized manner dictated in the usage model decided by the manufacturer.

PC Bootstrapping

- x86-based PC's start at memory location 0xFFFFFFF0 which is flash memory (BIOS)
  - BIOS verifies its integrity.
  - runs initialization routines provided by video card and disk controller.
  - Memory Test.
  - Assign interrupt vectors to I/O devices. Allow user to enter setup mode.
  - Look for the bootable drive; load its boot sector into memory. Jump to boot sector.

Boot Sector

- Initialize stack.
- Load kernel from disk into memory.
- Jump to kernel.
  - and so on: switch to protected mode, set up memory management, interrupt handling, file-system, input devices, etc.

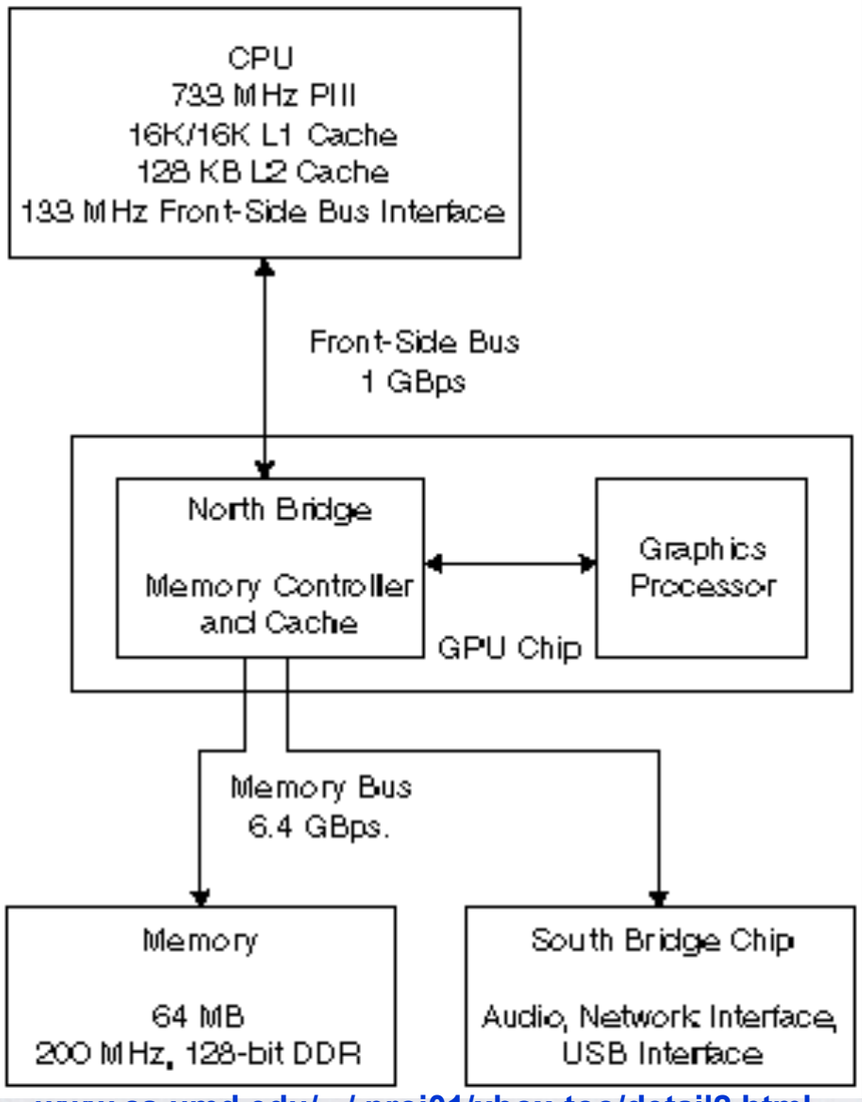
XBox bootstrapping

- Use of BIOS in a flash chip or similar is not the best solution for startup:
  - such chips can be replaced or overridden by adding another chip on the bus (such functionality is useful at the manufacturing process and by default remains in the final product).
  - contents of flash chips may be overwritten.

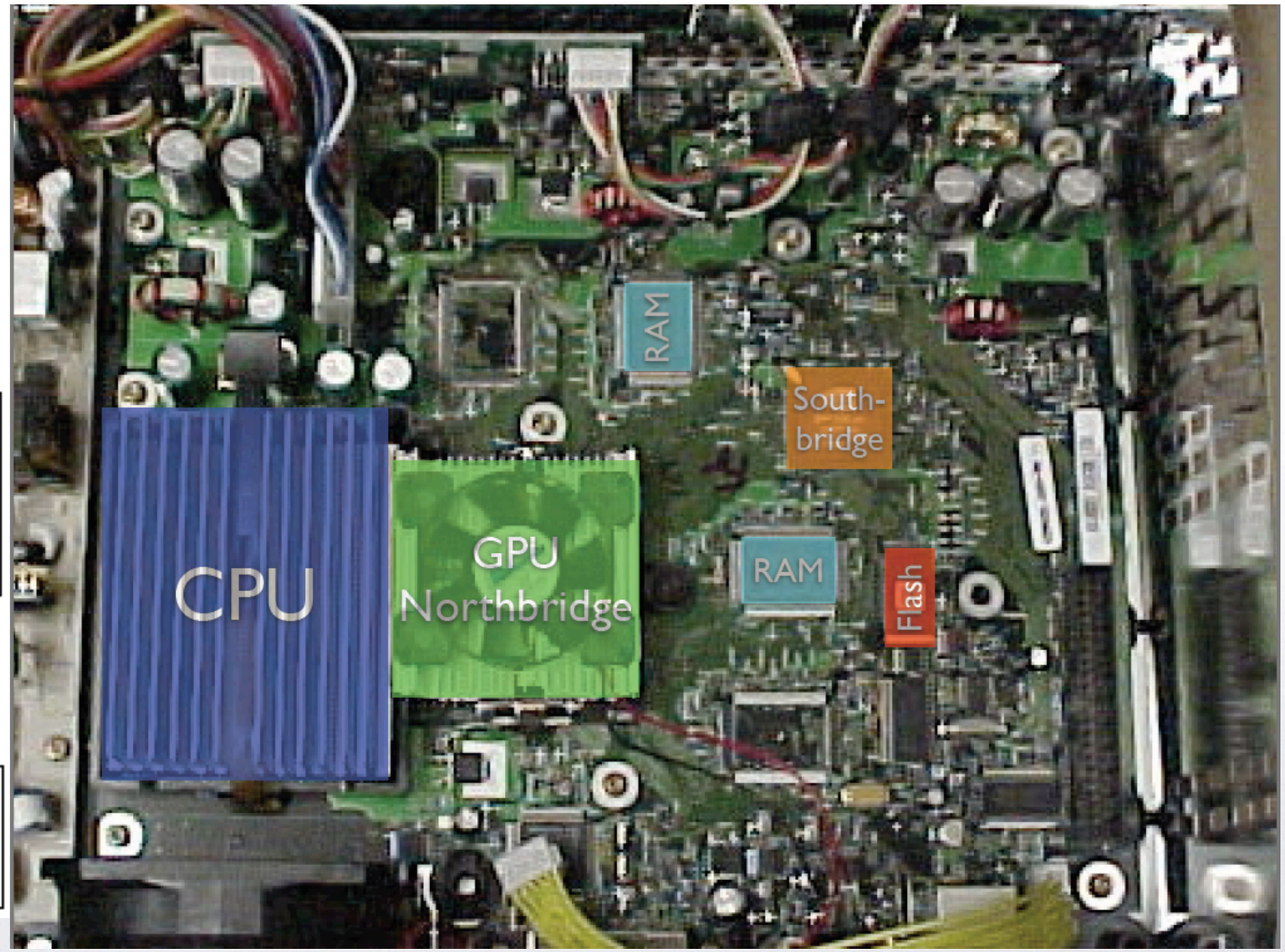
XBox solution

- Use a tiny hidden ROM:
  - it will be executed first.
  - will load rest of firmware data from flash.
  - the hidden ROM will perform integrity check on the flash contents.
- Where to put it? what would be the best solution? A separate chip? Inside the CPU?

XBox architecture



www.cs.umd.edu/.../ proj01/xbox-too/detail2.html



XBox Bootstrapping

- On reset, execute tiny hidden ROM code (512 bytes).
- Such code may load and verify the Windows Kernel that is stored in flash memory (1MB). *How to verify?*
  - use a hash (e.g. SHA-1 of the Kernel)?
  - use an RSA digital signature?
  - use a MAC?

Use a Second Bootloader

- Use the hashing approach.
- Instead of the Kernel have the BIOS verify *a second bootloader* (that would never change).
- The 2bl would verify the integrity of the Kernel using strong cryptographic integrity checking (RSA signatures).
- But is it a good idea to have the code of the 2bl + Kernel in plaintext?

Encrypted Flash

- XBox flash memory is encrypted.
  - encryption puts forth key-management issues: where to store the key?
- Only reasonable place: the tiny secret-ROM. Together with the decryption algorithm? (remember only 512 bytes available).

Is decryption easy?

- XBox RAM can be unstable; RAM initialization (test & clock the RAM) should occur prior to doing any meaningful calculations.
- RAM initialization, data decryption + hashing in 512 bytes? possibly at least 2KB.
- Outsource parts of the code to Flash?

Virtual Machine

- Store an interpreter using a small library of functions into the secret ROM (i.e., a essentially a special purpose opcode language to run the machine).
- Store the actual code for doing the startup operations in the clear within the Flash chip.
  - challenge: make sure that the attacker cannot gain any substantial advantage by modifying interpreted code (it was dubbed xcode).
- Interpreter ~ 175 bytes.

XCodes

- XCodes can do the following:
  - read memory and access I/O ports. Necessary to do the tests etc.
  - But then what prevents an attacker from writing xcode that prints the contents of the hidden ROM? Interpreter makes sure such reads are not allowed.
  - Other similar controls are added.

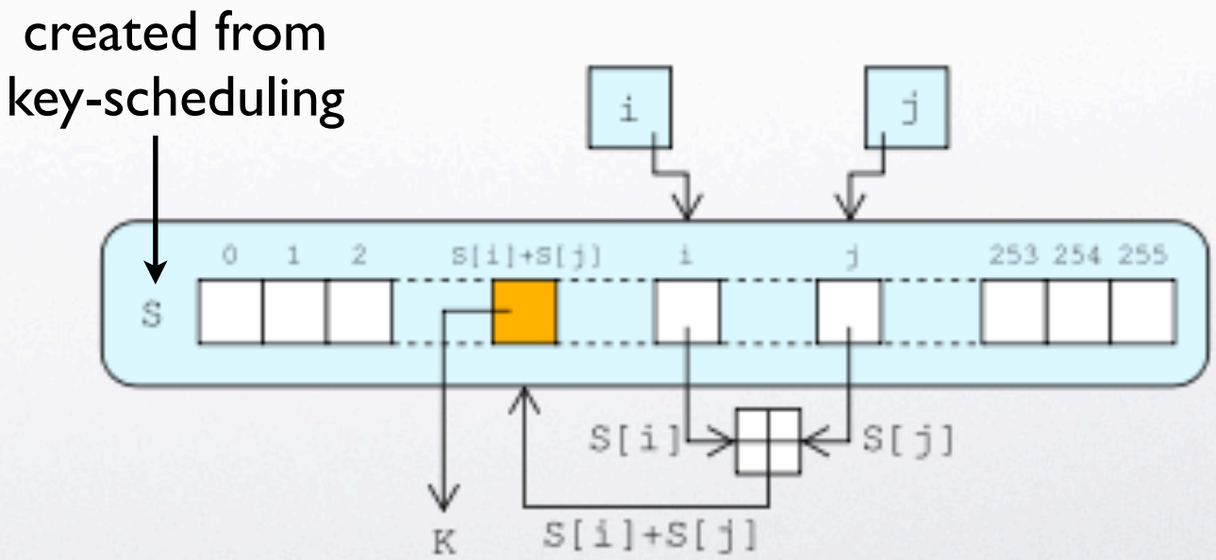
Encryption of 2BL

- RC4 algorithm -- fits into 150bytes using a 16 byte key (stored in the secret ROM of course).

RC4

- (Rivest Cipher 4).
- A stream cipher. one-time pad paradigm using pseudorandom sequence of bits.

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap(S[i],S[j])
    output S[(S[i] + S[j]) mod 256]
endwhile
```



Running out of Space?

- The interpreter + RC4 implementation were filling up the 500bytes quickly.
- No space for integrity check... so the final four bytes resulting from the RC4 implementation was used for integrity checking: (compare state to “0x7854794A”, if same continue else panic!)
- A good idea?

How to panic?

- If the machine simply stops and flashes in error an attacker may be capable of dumping the hidden ROM contents.
- If the machine totally shuts down is not user-friendly.
- Solution? partial shut down: turn off the secret ROM (that should be possible anyway after a successful transfer of control to the 2bl).
- But how to do simultaneously hlt the CPU and shut down the hidden ROM contents from within the hidden ROM? *Solution: turn off the secret ROM at the very last instruction of the address space; this overflows the EIP register causing an unhandled exception which effectively halts the machine!*

Hacker's perspective

- Knew some Xbox chipset details. Found out some more info on filesystem and that there was an apparent lack of any kernel data into the disk space.
- But the Xbox Dashboard was found in one of the partitions.

Desoldering

- Andrew "bunnie" Huang, disassembled his Xbox, desoldered the flash memory and put its contents on his web-site.

- Guess what happened next?
- Amidst the encrypted nonsense he observed readable x86 code in the upper 512 bytes of the flash dump. *Code that was not used as it was found out by reprogramming the flash...*

The mystery code

- It was a previous version of the hidden ROM code that was mistakenly linked into the flash during production!

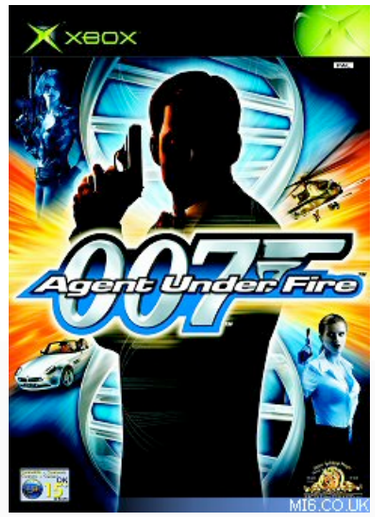
- Still, knowing that some other code gets written there, the hidden code was sniffed out of the bus that connects the southbridge to the CPU.

- It was found immediately afterwards that doing an integrity check with RC4 was not very smart!

The result

- With the integrity check defeated and knowledge of all secret Xbox ROM code it was possible to make modchips that could make Xbox do anything.
- Microsoft responded by updating the ROM and installing a better integrity check.
  - ... that took two weeks to hack!
    - reason: their developers seem to not know much about cryptography; neither about the hardware used!

Game Exploits



- The above give you total control with a little bit of hardware interaction...
- But Buffer overflows in commercial games delivered full control at the software level! (i.e., you could boot whatever you want without opening the Xbox).

Lessons learned

- Security vs. money / Security vs. speed / Hacker's resources / Security through obscurity / leftover code .
- combinations of weaknesses; composability of partially faulty components:
  - boot process vulnerable so games were analyzed.
  - save games buffer overflow.
  - games run in kernel mode.
  - dashboard does not verify the integrity of font files
  - font file buffer overflow
  - hacker's resources.

Conclusion for XBOX

- "The security system of the Xbox has been a complete failure"
- Read: 17 Mistakes Microsoft Made in the Xbox security system, Michael Steil, Xbox Linux project.

XBox 360

- Three-core IBM PowerPC based, 3.2 GHz.
- 1MB L2 cache
- 512MB RAM
- ATI Graphics processor 10MB.
- 20GB IDE Hard Disk/ IDE dvd
- Fast Ethernet / USB / Wifi ready
- O/S derived from Windows 2000 Kernel.

XBox 360

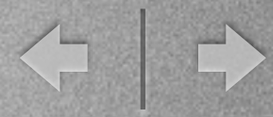
- Overview of problems that the new XBox had to deal with:
  - Bus Sniffing
  - Flash reprogramming
  - Manufacture-time override
  - DVD attack
  - Code injection via DMA

Build in the CPU

- Put boot ROM + temp RAM for keys and work inside the CPU.
- takes care of issues like bus sniffing and flash reprogramming.

Separation of Roles

- Games should not run in Kernel mode (Hypervisor mode).
  - It enforces W^X everywhere (every page in process space should be either writeable or executable but not both).
  - Game privileges exclude loading executable code.

Prevent DMA injections

- Everything that goes out of the cache of CPU is hashed and kept internally.
- When loaded the hash is checked.
- If a device "goes wild" and overwrites something in RAM the hash will (hopefully) not match.
- Any hash mismatch makes the CPU to lock.

Issues

- DMA is needed (how would you load things from drives?).
- CPU-internal RAM is a premium
  - cache line 128 bytes; hash is 16 bytes
  - with 64KB we can secure 1MB of RAM.

Using Encryption

- Have everything encrypted in RAM.
- Encryption / Decryption works at the Cache level.
- In this way any direct RAM modification (that is going outside of Hypervisor) will be unable to inject useful data.
  - As long as Hypervisor stands it is good protection

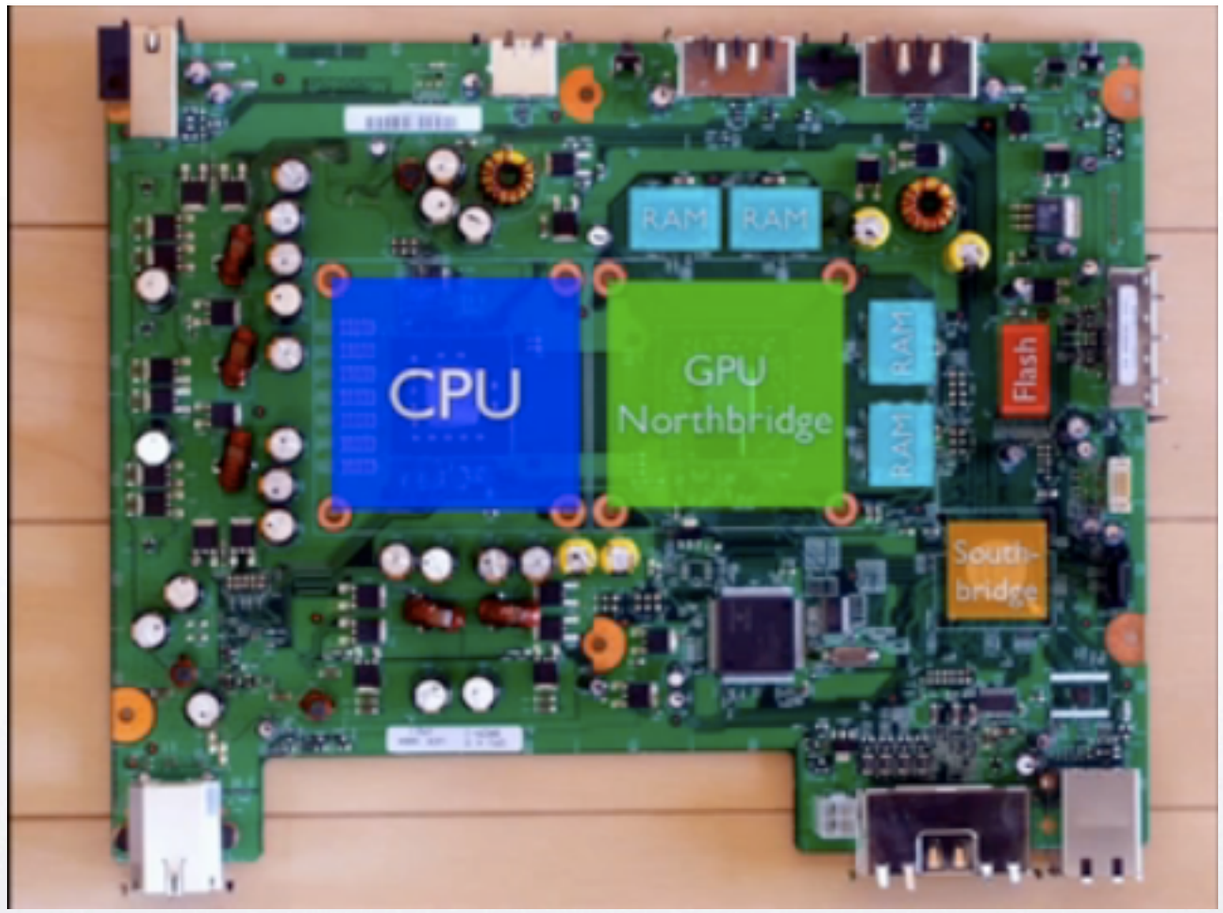
Using fresh keys

- By building a RNG into the CPU:
  - we can use a different key each time the system boots for all memory encryptions.
  - prevents replay attacks.

eFUSEs

- Dynamic real-time reprogramming of CPU
  - A chip can change while in operation -- in a controlled way.
  - "Blowing" an efuse allows
    - to disable verification / testing features.
    - build in a key.
    - count updates.

XBox 360 Internals



768 bit
eFUSEs

768 bit
eFUSEs

XBOX 360 Startup

- At manufacture time a generic image boots
  - all eFUSEs are set to 0.
- Unique key is generated and stored.
- DVD key is also stored.
- after terminating any external access is shut-down (by blowing suitable eFUSEs).

XBOX 360 boot

- Hypervisor loads.
- Check kernel code (w2000 derived) for signatures.
- Kernel can load code with the help of the Hypervisor only.
- Updates are sequenced.

the bug

64 bit

The Hypervisor Syscall Handler

Disassembly

```
13D8: cmplwi %r0, 0x61
13DC: bge illegal_syscall
...
13F0: rldicr %r1, $r0, 2, 61
13F4: lwz $r4, sc_table(%r1)
13F8: mtlr %r4
...
1414: blrl
```

Pseudo Code

```
handle_syscall(syscall_nr,arg0,arg1,...) {
    if (syscall_nr >= 0x61)
    goto illegal_syscall;
...
    extern long sc_table[0x61];
    void (*ptr)() = sc_table[syscall_nr];
ptr(arg0, arg1, ...);
...
}
```

32 bit

The high 32 bits are used by the CPU for flags (e.g., encryption/decryption)
By tampering with syscall we can switch off the encryption flag!

=> If we can store some code (unencrypted) at the syscall address it could be executed


Games and Shader code

- *Shader* code is not signed, is executable by the GPU and has access to memory.
- So if we make appropriate shader code it can plant the malicious code for the modified syscall to run.
- But how do we make the actual syscall that will run our code?

Running the code

- Thread switching by CPU requires saving registers.
- Turns out : these are saved unencrypted.
- So we can overwrite these registers as well and put the program counter where we want it!

What went wrong?

- Hypervisor bug.
- Run-time register values are very critical and they were left unauthenticated (performance trade-offs)
- why should hypervisor circumvent the MMU.

Historical

device	y	security	hacked	for	effect
PS2	1999	?	?	piracy	-
dbox2	2000	signed kernel	3 months	Linux	pay TV decoding
GameCube	2001	encrypted boot	12 months	Homebrew	piracy
Xbox	2001	encrypted/signed bootup, signed executables	4 months	Linux Homebrew	piracy
DS	2004	signed/encrypted executables	6 months	Homebrew	piracy
PSP	2004	signed bootup/executables	2 months	Homebrew	piracy
Xbox 360	2005	encrypted/signed bootup,encrypted/signed executables, encrypted RAM, hypervisor, eFuses	12 months	Linux Homebrew	leaked keys
Wii	2006	encrypted bootup	1 month	Linux	piracy
PS3	2006	encrypted/signed bootup,encrypted/signed executables, hypervisor, eFuses, isolated SPU	not yet	-	-

From :
Felix Domke
Michael Steil
GoogleTech Talk.