TRESOR Runs Encryption
Securely Outside RAM

20th USENIX Security Symposium
August 8 – 12, 2011 • San Francisco, CA

Tilo Müller    Felix C. Freiling    Andreas Dewald
Department of Computer Science
University of Erlangen
Who we are

Chair for IT-Security Infrastructures
University of Erlangen-Nuremberg
www1.cs.fau.de

Nuremberg
Franconia, Germany
PART I

Introduction
Motivation

Cold Boot Attacks

Firewire Attacks

Other DMA Attacks
  - PCI
  - PC-Card
  - Thunderbolt?

→ RAM is insecure
→ **Disk encryption** which stores the key in RAM is insecure

Affected: BitLocker, FileVault, dm-crypt, TrueCrypt and more
TRESOR's Security Policy

TRESOR Runs Encryption Securely Outside RAM:
- AES implementation solely on the microprocessor
- Secret keys and states never enter RAM
- Instead, only processor registers are used as storage
PART II

Implementation

/* generate next round key (128- and 256-bit) */
.macro key_schedule r0 r1 r2 rcon
+   pxor  rhelp, rhelp
+   movdqu \r0, \r2
+   shufps $0x1f, \r2, rhelp
+   pxor  rhelp, \r2
+   shufps $0x0c, \r2, rhelp
+   pxor  rhelp, \r2
+   aeskeygenassist $\rcon, \r1, rhelp
+ .if (\rcon == 0)
+   shufps $0xaa, rhelp, rhelp
+ .else
+   shufps $0xff, rhelp, rhelp
+ .endif
+   pxor  rhelp, \r2
+.endm
Key management: key storage

The key registers must be:

- big enough to store AES-128/192/256 keys (size)
- a privileged ring-0 resource (security)
- seldom used by applications and compensable in software (compatibility)

→ fulfilled by the set of debug registers
Key management: debug regs

TRESOR (mis)uses debug registers as persistent key storage

→ supports AES-128/192/256 on 64-bit machines
→ supports AES-128 on 32-bit machines
Key management: key derivation

```
>> TRESOR <<
Enter password  > ************
Confirm key hash > 71 47 15 e1 00 db 94 38 1a 38 fb 91 6f 2a ca 6e
                  42 ec 90 14 a0 9d fc 5c b8 b5 63 9b 4b c2 35 5e
Correct (yes/no) > yes
```
AES Algorithm: guideline

Security Policy: No valuable information about the AES key or state should be visible in RAM at any time

Challenge: Implement AES without using RAM at all

→ no runtime variables in data segment (stack, heap, …)
→ use SSE registers and GPRs to store intermediate states
→ written in assembly language (x86)
AES Algorithm: assembly implementation

1. Generic x86 assembler instructions → possible, but far too slow

2. Intel's new AES instruction set (AES-NI)
   - hardware accelerated AES instructions
     aesenc, aesenclast, aesdec, aesdeclast
   - runs without RAM (instead: SSE)
   - short and efficient AES code
→ does perfectly meet our needs

/* Encrypt */
.macro encrypt_block rounds
  movdqu 0(%rsi),rstate
  read_key rk0 rk1 \rounds
  pxor rk0,rstate
  generate_rks_\rounds
  aesenc rk1,rstate
  aesenc rk2,rstate
  aesenc rk3,rstate
  aesenc rk4,rstate
  aesenc rk5,rstate
  aesenc rk6,rstate
  aesenc rk7,rstate
  aesenc rk8,rstate
  aesenc rk9,rstate
  .if (\rounds > 10)
  aesenc rk10,rstate
  aesenc rk11,rstate
  .endif
  .if (\rounds > 12)
  aesenc rk12,rstate
  aesenc rk13,rstate
  .endif
  aesenclast rk\rounds,rstate
.epilog
.endm

/* Decrypt */
.macro decrypt_block rounds
  movdqu 0(%rsi),rstate
  read_key rk0 rk1 \rounds
  generate_rks_\rounds
  pxor rk\rounds,rstate
  .if (\rounds > 12)
  read_key rk0,rk1,10
  aesdec rk13,rstate
  aesdec rk12,rstate
  .endif
  .if (\rounds > 10)
  aesdec rk11,rstate
  aesdec rk10,rstate
  .endif
  aesdec rk9,rstate
  aesdec rk8,rstate
  aesdec rk7,rstate
  aesdec rk6,rstate
  aesdec rk5,rstate
  aesdec rk4,rstate
  aesdec rk3,rstate
  aesdec rk2,rstate
  aesdec rk1,rstate
  aesdeclast rk0,rstate
.epilog
.endm
AES Algorithm: key schedule

Conventional AES:
round keys are calculated once and then stored in RAM
(for performance reasons)

TRESOR:
on-the-fly round key generation
(since the entire key schedule is too big to be stored inside CPU)
Kernel Patch

We have to _patch the operating system kernel_ for two reasons:

1. Problem: unprivileged user access to debug registers → Solution: _patch ptrace_ syscall

2. Problem: scheduling and context switching of SSE /GPRs → Solution: introduce _atomicity_

Hence, TRESOR is implemented in kernel space (currently Linux 2.6.36)
Kernel Patch: key protection

Risks:
1. Malicious user access to debug registers
   → compromised key
2. Writing to debug registers accidentally (e.g., starting gdb)
   → polluting key storage
   → data corruption

Solution:
deny access to debug registers from userland

```c
int ptrace_set_debugreg (tsk_struct *t, int n, long v)
{
    thread_struct *thread = &(t->thread);
    int rc = 0;
    if (n == 4 || n == 5)
        return -EIO;
    if (n == 6) {
        thread->debugreg6 = v;
        goto ret_path;
    }
    if (n < HBP_NUM) {
        rc = ptrace_set_breakpoint_addr(t, n, v);
        if (rc) return rc;
    }
    [...
    ret_path: return rc;
}
```
OS regularly performs CPU context switches
when TRESOR is active this context comprises sensitive data
(general purpose and SSE registers)

⇒ run TRESOR atomically
(per 128-bit input block)

```c
/* Encrypt one TRESOR block */
void tresor_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
    struct crypto_aes_ctx *ctx = crypto_tfm_ctx(tfm);
    unsigned long irq_flags;

    // enter atomicity
    preempt_disable();
    local_irq_save(*irq_flags);

    // encrypt block
    switch(ctx->key_length) {
        case AES_KEYSIZE_128: tresor_encblk_128(dst,src); break;
        case AES_KEYSIZE_192: tresor_encblk_192(dst,src); break;
        case AES_KEYSIZE_256: tresor_encblk_256(dst,src); break;
    }

    // leave atomicity
    local_irq_restore(*irq_flags);
    preempt_enable();
}
```
PART III

Security Evaluation
Security Analysis: memory attacks

TRESOR: nothing but the output block is written actively to RAM

But: sensitive data may be copied into RAM passively by OS side effects (e.g., interrupt handling, scheduling, swapping, ACPI suspend, etc.)

→ observe RAM of a TRESOR system at runtime

Test-Setup:
- KVM/Qemu
- guest1: unpatched Linux, no encryption
- guest2: unpatched Linux, generic AES encryption
- guest3: patched Linux, TRESOR encryption
- examine guests main memories from the host
Security Analysis: memory attacks

Test 1: Browse guest's main memory with AESKeyFind.

Result:
- guest 1 (no enc): no key recovered
- guest 2 (generic AES): key recovered
- guest 3 (TRESOR): no key recovered

But:
AESKeyFind is heavily based on the AES key schedule. Since TRESOR does not store a key schedule, this may be the only reason why the key cannot be recovered.
Security Analysis: memory attacks

Test 2: Unlike real attackers we are aware of the secret key. → we don't need the key schedule but can search for the key bit pattern directly.

Result:
- guest 1 (no enc): /-
- guest 2 (generic AES): match found
- guest 3 (TRESOR): no match found

But:
The key could be stored discontinuously, in another endianess, etc.
Security Analysis: memory attacks

**Test 3:** Search for the longest match of the key pattern, its reverse and any part of those, in little and in big endian.

**Result:**
- guest 1 (no enc):    -/-
- guest 2 (generic AES): 32-byte longest match
- guest 3 (TRESOR):  3-byte longest match

**But:**
The key could enter RAM only seldom, in special situations.
Security Analysis: memory attacks

Test 4: Search for the longest match of the key pattern during ACPI suspend and during swapping.

Result (suspend-to-RAM):
- guest 2 (generic AES): 32-byte longest match
- guest 3 (TRESOR): 3-byte longest match

Result (swapping):
- guest 2 (generic AES): 3-byte longest match on disk
- guest 3 (TRESOR): 3-byte longest match on disk

But:
These are only the most important special states of the Linux kernel. Unfortunately, it is practically impossible to put the Linux kernel into all its different states and analyze its memory at the right moment.
**Security Analysis: memory attacks**

**Test Summary:**

<table>
<thead>
<tr>
<th>AES variant:</th>
<th>Generic AES</th>
<th>TRESOR</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel state:</td>
<td>normal</td>
<td>normal</td>
<td>swapping</td>
</tr>
<tr>
<td>AESKeyFind</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Exact key match</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Longest match</td>
<td>32 bytes</td>
<td>3 bytes</td>
<td>3 bytes</td>
</tr>
</tbody>
</table>

→ we never found sensitive information in RAM or on disk
Security Analysis: processor attacks

Cold Boot Register Attack

- Virtual Machines (tested on Qemu, Boch, Vmware and VirtualBox) vulnerable
- Real Hardware (tested on seven different CPUs and BIOS versions) not vulnerable
Security Analysis: processor attacks

Compromise system space

```bash
insmod picklock.ko ; dmesg | tail -n 28
```

```
[240512.336708] =====================
[240512.336711] DEBUG REGISTERS:
[240512.336841]
[240512.336843] CPU 0
[240512.336846] db0: 0xc7084b3286a3c6eb
[240512.336850] db1: 0xe33d5a7a5db2aa66
[240512.336853] db2: 0xc4e27ee4fea598e2
[240512.336856] db3: 0xff10831b4cbca50b
[240512.337172]
[240512.337173] CPU 1
[240512.337176] db0: 0xc7084b3286a3c6eb
[240512.337179] db1: 0xe33d5a7a5db2aa66
[240512.337181] db2: 0xc4e27ee4fea598e2
[240512.337184] db3: 0xff10831b4cbca50b
[240512.337249]
```

Always possible with superuser rights if
- LKMs are supported
- or /dev/kmem can be written
PART IV

Future Work
Currently TRESOR supports …

- AES-128 on 32-bit machines
- AES-128/192/256 for 64-bit/AES-NI machines
- multi core/processor environments
- hibernation / suspend-to-RAM
- kernel level encryption: dm-crypt
- Linux kernel 2.6.36
Future Work

Upcoming releases of TRESOR will support …

- multiple keys and session keys
  (holding a master-key-encrypted keyring in RAM)

- userland encryption
  (via syscalls or, better, via sysfs)

- optionally MSRs instead of debug registers
  (to restore ability of hw breakpoints on a chosen set of CPUs)

- sealing the symmetric key by TPM
  (like BitLocker)

- runtime management
  (enable/disable TRESOR, set new key at runtime, etc.; a bit more insecure but required by server systems with remote-access only)

- Linux kernel 3.0
  (and more long-term stable releases from there on)
TRESOR's name

btw: TRESOR is not just another recursive backronym, it's German for safe / vault ;)

<table>
<thead>
<tr>
<th>Substantive (5 of 5)</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>safe</td>
<td>Tresor</td>
</tr>
<tr>
<td>security container</td>
<td>Tresor</td>
</tr>
<tr>
<td>strong room</td>
<td>Tresor</td>
</tr>
<tr>
<td>strongbox</td>
<td>Tresor</td>
</tr>
<tr>
<td>vault [bank.]</td>
<td>Tresor</td>
</tr>
</tbody>
</table>
Thank you for your attention.

Questions?

E.g., Do you publish the source code?
Of course, it's available under GPLv2 here:
www1.cs.fau.de/tresor