Blinded Random Corruption Attacks

Rodrigo Rubira Branco (@BSDaemon)
Principal Security Researcher
rodrigo.branco *noSPAM* intel.com

Shay Gueron (Core Architecture and Haifa University)
Senior Principal Engineer

DISCLAIMER

We don’t speak for our employer. All the opinions and information here are of our responsibilities

– So, mistakes and bad jokes are all

– **OUR** responsibilities

Agenda

- Introduction
- The A-B-C Model for the Attacker’s Level of Control and Targets
- Becoming “root” on a locked system with a BRBC attack
- Different attack scenarios and attack targets
- Mitigation Techniques
- Conclusions

Background

- Adversaries with physical access to attacked platform – are a concern
  - Mobile devices (stolen/lost)
  - Cloud computing (un-trusted environments)
- Read/write memory capabilities as an attack tool have been demonstrated:
  - Using different physical interfaces
  - Thunderbolt, Firewire, PCIe, PCMCIA and new USB standards
- Consequences of DRAM modification capabilities:
  - Active attacks on memory are possible
  - Attacker can change code / data **from any value to any chosen value**

Underlying attack assumption on the threat model:
The attacker has physical means to modify DRAM

Different attacker tactics

- Passive attack: the attacker can only eavesdrop DRAM contents, but is not able to inject or interfere with it (in-use or not)
  - Non-existent in reality

- Active static attack: the attacker can read DRAM contents but cannot modify in-use/to-be-used (saved) DRAM
  - Example: cold boot attack
  - The attack is on the data privacy

- Active dynamic attacks: the attacker can read and modify DRAM contents that are in-use/to-be-used (saved)

Memory Encryption without Authentication effectiveness limited to active static attacks since the ability to modify in-use/to-be-used DRAM is denied

Introduction (1)

- Some memory protection technologies against active dynamic attacks were proposed
  - Limiting the attacker's physical ability to read/write memory
    - E.g., blocking DMA access in some scenarios
  - Memory encryption
- **Memory encryption using "transparent encryption" mode:**
  - Simpler, cheaper, faster than "encryption + authentication"
  - Changes the assumptions on read/written memory capabilities of the attacker
  - Therefore, seems to be effective for limiting active dynamic attacks
- Memory encryption effects :
  - Attacker has **limited control** on the result of active attacks
  - But the physical memory modification **capabilities remain available**

Underlying attack assumption: attacker has physical means to modify DRAM

Introduction (2)

- Under memory encryption, the attacker has limited capabilities
    - **B**linded **R**andom **B**lock **C**orruption (**BRBC**) attack
- (**Blinded**) The attacker does not know the plaintext memory values he can read from the (encrypted) memory.
- (**Random** (**Block**) **Corruption**) The attacker cannot control nor predict the plaintext value that would infiltrate the system when a modified (encrypted) DRAM value is read in and decrypted.
    - When using a block cipher (in standard mode of operation), any change in the ciphertext would **randomly corrupt** at least **one block** of the eventually decrypted plaintext
- The question: can memory encryption (that limits the active dynamic attacker capabilities to **BRBC** only) provide a "good enough" mitigation in practice?

Underlying attack assumption: attacker has physical means to modify DRAM

Introduction (3)

- We will show that:
  - Despite limited capabilities dynamic active attacks are still possible
  - Encryption-only does not offer a defense-in-depth mechanism against arbitrary memory overwrites without removing capabilities assumptions
- The BRBC attacker is able to create Time-of-check/Time-of-use (TOCTOU) race conditions all around the execution environment
  - Usual control-flow hijacking attacks require precise pointer control to redirect flow of execution. Usual DMA attacks perform precise code modification
  - Data-only attacks caused by a BRBC attacker can be induced after some code checks, therefore cause TOCTOU races that invalidate the results of such checks
  - Unexpected computation (and flows) can emerge (since code is driven by its input data)
    - Data-only based attacks, thus control flow enforcement can't prevent

Underlying attack assumption: attacker has physical means to modify DRAM

The A-B-C attacker model

- **A**ccess Seeking Attacker

This attacker is not the owner of the platform, but got it to his possession, in a locked state. He wishes to get an user access, in order to steal the data on the system.

- **B**reaching Attacker

This attacker is a legitimate user of the platform, who wishes to breach some of the system’s policies or circumvent restrictions on his privileges.

- **C**onspirator Attacker

This attacker is also a legitimate user of the platform/environment. He has administrative powers and conspires to collect other users’ data.

Underlying attack assumption: attacker has physical means to modify DRAM

Becoming “root” on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related…
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming “root” on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related…
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming “root” on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related…
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming "root" on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related…
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming “root” on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related…
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming "root" on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related…
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming “root” on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related…
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

BRBC Attack to the preauth_flag

Becoming "root" on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related…
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();         -> THIS NEVER GETS EXECUTED!

        auth_ok:
                return;
}
```

TOCTOU (Time-of-use/Time-of-check) Race Condition

- This was caused by our arbitrary memory write (the BRBC)
- The corrupted values adjacent to the preauth_flag were not used at this moment (thus the block corruption is not a problem)
- The check for the preauth_flag only checks for not 0 (thus we don't need to control the exact value)

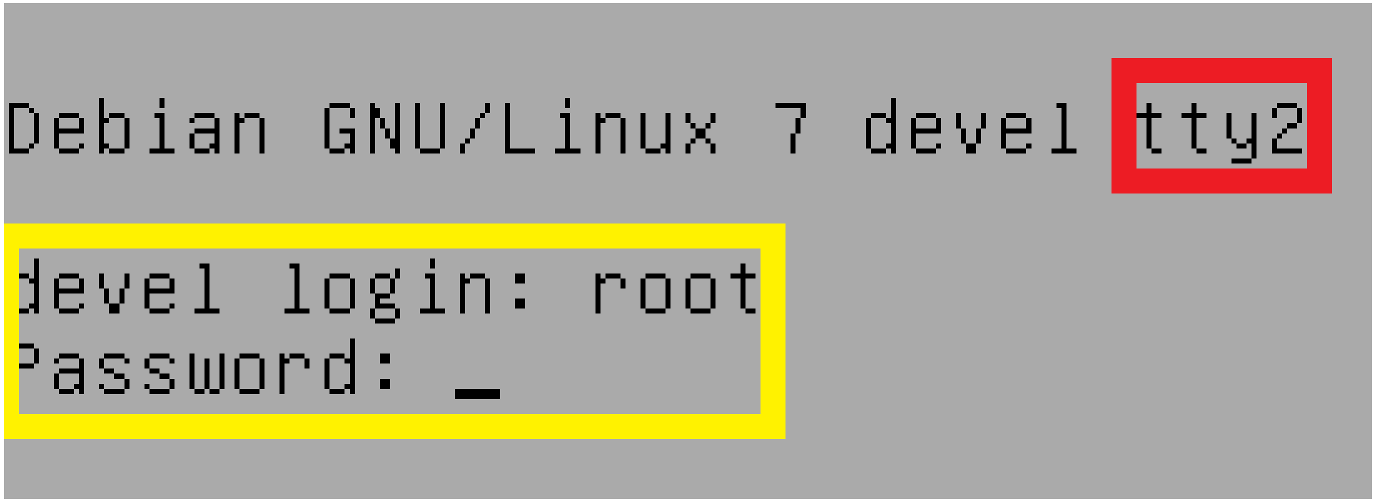
- But how do we win the race?
  - In this case, quite simple: We just cause the authentication to fail at the first time (when it does ask the password)
    - The system waits for the password prompt
    - We cause the corruption and input invalid password
    - The authentication fails and the logic is repeated, but this time with the corruption!

Experiment

- The demonstration was created focusing on the underlying attack assumption
  - We use a debugger to make it easy to step through and see the corruption effect
  - We use the JTAG to demonstrate the physical addresses are not a concern
- SW mitigations are not feasible because the attacker has lots of possibilities for targets (not only ! 0 comparisons). Some examples:
  - If an attacker overwrites the NULL terminator of a string, he can generate buffer overflows, memory leaks
  - If an attacker overwrites an index, he can generate out-of-bounds writes, that might lead to user-mode dereferences if in kernel-mode context
  - If an attacker overwrites a counter, he can generate REFCOUNT overflows, leading to use-after-free conditions

Underlying attack assumption: attacker has physical means to modify DRAM

Attack Demonstration Using a Debugger



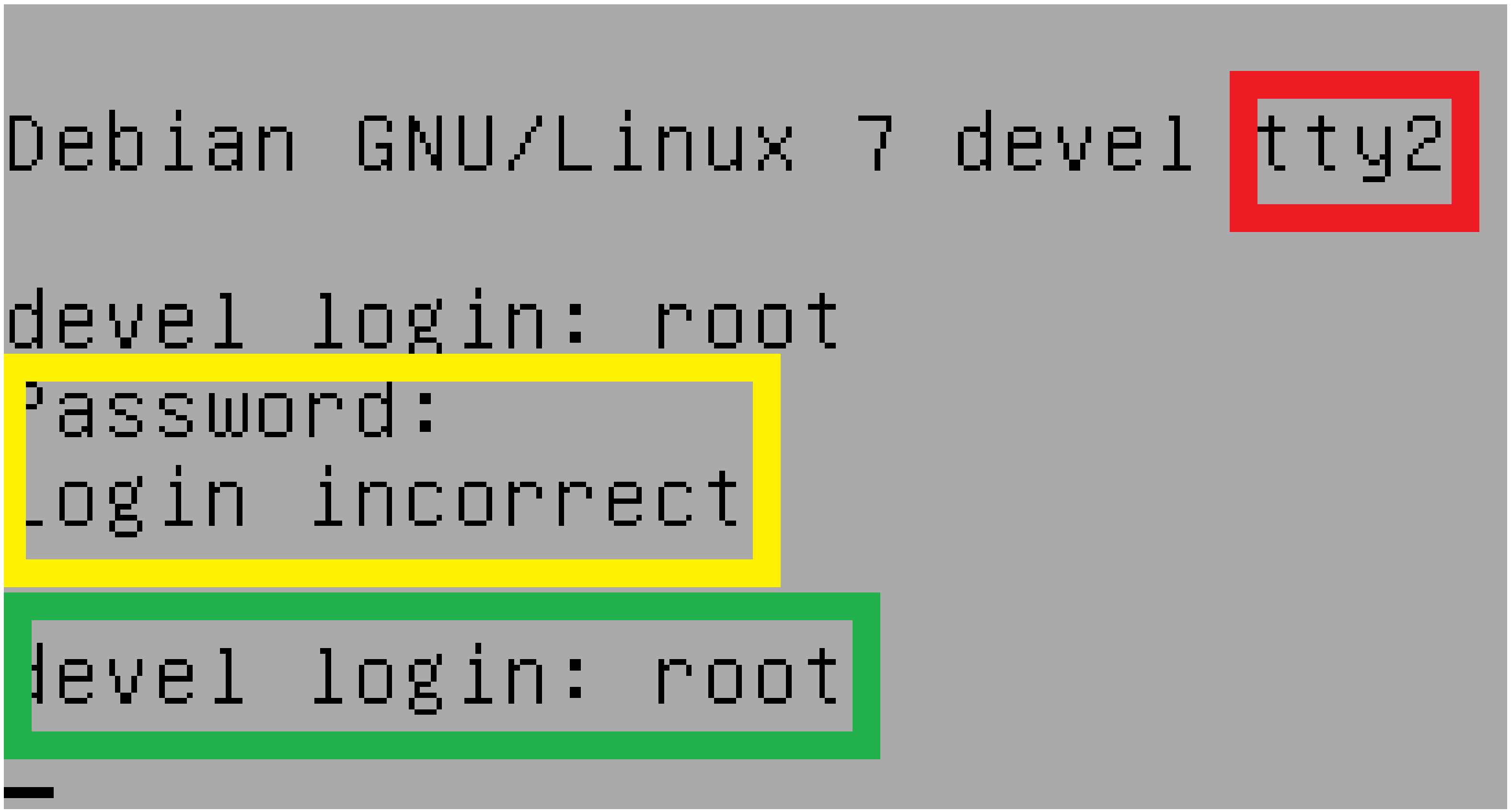
Attack Demonstration Using a Debugger

```
root(tty1)@devel:~/Shay# ps ax |grep login
 2987 tty2      Ss+    0:00 /bin/login --
 2989 tty1      S+     0:00 grep login
root(tty1)@devel:~/Shay# gdb /bin/login 2987
GNU gdb (GDB) 7.4.1-debian
```

Attack Demonstration Using a Debugger

```
0xb77cf424 in __kernel_vsyscall ()
(gdb) b *0x804a6e6
Breakpoint 1 at 0x804a6e6: file login.c, line 966.
(gdb) c
Continuing.
```

Attack Demonstration Using a Debugger



Attack Demonstration Using a Debugger

```
0xb7711424 in __kernel_vsyscall ()
(gdb) b *0x804a6e6
Breakpoint 1 at 0x804a6e6: file login.c, line 966.
(gdb) c
Continuing.

Breakpoint 1, 0x0804a6e6 in main (argc=3, argv=0xbfe82554) at login.c:966
966                                 spwd = xgetspnam (username);
(gdb) set preauth_flag="16 bytes garbage"
(gdb) c
Continuing.
process 2987 is executing new program: /bin/bash
```

Attack Demonstration Using a Debugger

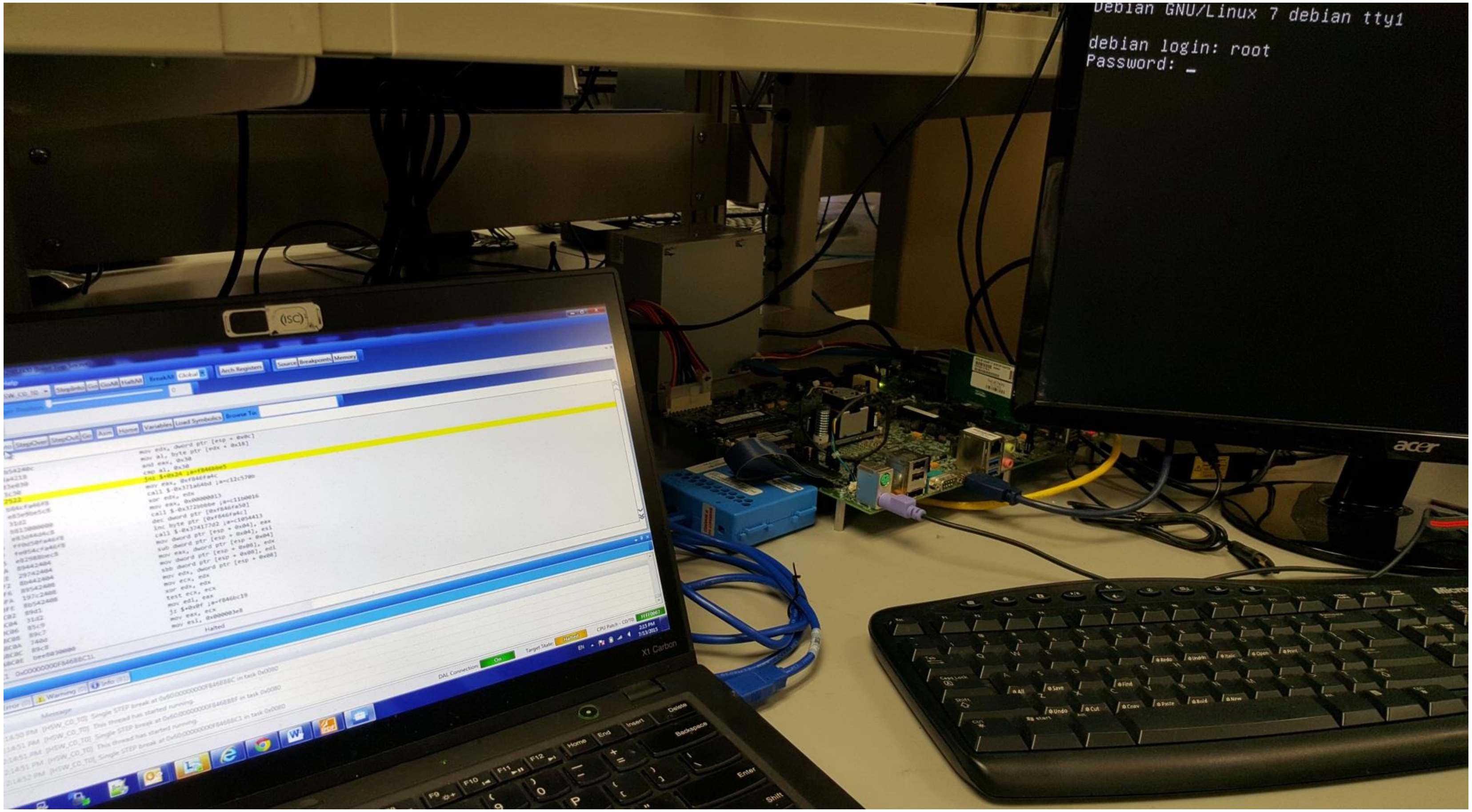
```
Debian GNU/Linux 7 devel tty2

devel login: root
Password:
Login incorrect

devel login: root
1 failure since last login.
Last was Mon Mar  9 11:31:26 2015 on /dev/tty2.
root(tty2)@devel:~# whoami
root
root(tty2)@devel:~# id
uid=0(root) gid=0(root) groups=0(root)
root(tty2)@devel:~# _
```

Attack Demonstration using the JTAG Interface

- The difference on the JTAG demonstration is:
  - Establish the possibility of the attack against the physical address space instead of the virtual one (as with the debugger)
  - Demonstrate that blinded reads are enough to gather locality of the targeted overwrite
  - Understand possible mitigations and their impacts on the attack (for example, control-flow enforcement technologies would not have prevented the attack either and can’t be considered another layer of defense against BRBC)

- Limitations of the JTAG attack
  - For the MEE case, the JTAG access would be encrypted/decrypted, thus it would not be dealing with the encrypted content

Attack Demonstration using the JTAG Interface



Different Attack Scenarios and Attack Targets

- Attacker with user privileges on the machine
  - Higher control/visibility of the memory space
  - Tries to bypass security policies
    - Local administrator (common on cloud-based scenarios)

- All system software/components can be seem as targets
  - We just demonstrated in a highly-limited scenario (locked machine, unknown software running, little to no information on the OS details)

- As more interactions with the system, as bigger is the scope of possible attack targets (as discussed previously)

Mitigation Techniques

- Hibernation when used together with proper disk encryption

- VT-d/IOMMU and PMRs
  - Limits DMA capabilities exposed
  - Might not be enough against certain attackers (that have physical access) and in some platforms (only effective if the attack requirement is fully removed)

- Software self-protection (or control flow enforcement technologies)
  - Attack uses valid flows with invalid data (data-only attack) bypassing them
  - Different attack targets make software hardening inviable

- Memory encryption with Authentication
  - Able to detect the arbitrary change and prevent the attack

- Intel SGX (Software Guard eXtensions)
  - Currently employ authentication and replay protection

Conclusions

- Formalization of the BRBC attack
- Hierarchical model of the A-B-C attackers
- Practical demonstration of a BRBC attack
  – Definition of premises
  – Data-only attack generating a TOCTOU
- Discussion on mitigation paths

- **Encryption-only by itself is not necessarily a “good enough” defense-in-depth mechanism against arbitrary memory write primitive**

To consider

- Since encryption-only is not a defense against arbitrary writes (that not only break integrity, but also the confidentiality as demonstrated):
  - What is easier/viable: Remove ***ALL*** cases of arbitrary writes for ***ALL*** platforms the technology would support (which would depend on integration teams capabilities to guarantee that) or
  - At the technology level support encryption with authentication, which is a solution based only on itself

End! Really is !?

Rodrigo Rubira Branco (@BSDaemon)
Principal Security Researcher – Security Center of Excellence Core Client
rodrigo.branco *noSPAM* intel.com

Shay Gueron
Senior Principal Engineer - Core Architecture and Haifa University