

# Exploração Bizarra de Recursos Computacionais Incomuns em Software ou “Como criar um título randomicamente atraente”

Rodrigo Rubira Branco (@BSDaemon)  
Senior Principal Security Researcher  
Intel Corporation – Security Center of Excellence

# DISCLAIMER

**I don't speak for my employer. All the opinions and information here are of my responsibility**

- So, mistakes and bad jokes are all MY responsibility
- I do not use in this presentation a formal/academic description and instead try to make a pragmatic one for the discussion
- Yes, I use English and Portuguese all around, live with that 😊

# Créditos

- Esta apresentação conta com idéias, teorias, discussões, código, pesquisas das seguintes pessoas:
  - Sergey Bratus (Insecurity Theory, Exploiting the Hard-working Dwarf)
  - Meredith Patterson (Langsec)
  - R.I.P. Len Sassaman (Langsec)
  - James Oakley (Exploiting the Hard-working Dwarf, Katana)
  - PaX Team (memory corruptions and threat model for considering them)
  - Shay Gueron - Senior Principal Engineer, Amazon Web Services (AWS) (Ataques a sistemas com criptografia de memória)
  - Com certeza muitos outros!

# Agenda

- **Part I:** Into exploits, primitives and weird machines
- **Part II:** Exploiting Additional Computations
  - Past year's presentation wrap-up
- **Part III:** What is new for 2017?
  - How powerful primitives keep given
  - New classes/techniques for BRBC attacks

# Resumo

- Exploração de software deixou de ser genérica
- Primitivas de exploração em diferentes contextos
- Técnicas de exploração mostram como aproveitar primitivas
- Existe muito mais sendo 'computado' do que apenas o código propriamente escrito

# Bugs of Interest (kinds of exploits)

- Memory corruption bugs (\*)
  - Unintended control over address/content of memory access
    - “Precursor” bugs included (memory disclosure, unintended reads, others)

(\*) Definition taken from: “RAP: RIP ROP” Presentation @H2HC 2015 by PaX Team

# Threat Model (\*)

- Union of the powers (unintended side-effects) of all possible memory corruption bugs
- Arbitrary read-write memory access
- Bug can be triggered:
  - For arbitrary addresses
  - With arbitrary content
  - Arbitrary operation
  - Arbitrary number of times
  - At arbitrary times
- Consequences:
  - Privilege abuse: exercise existing powers for unintended purposes
  - Privilege escalation: gain new powers (to subsequently abuse them)

(\*) Taken from: “RAP: RIP ROP” Presentation @H2HC 2015 by PaX Team

# Prevenção não funciona?

- Iniciativas de ‘computação confiável’
- Vários livros de ‘programação segura’
- Diversas publicações acadêmicas
- Infinitudes de métodos de teste: fuzzing, análise de binários, etc
  - E o software continua sendo ruim E explorado!
- E o hardware (não Intel, claro)... nem sequer temos idéia de quão ruim ele é...



# Insegurança é sobre computabilidade

- *“Confiança de um sistema computacional é sobre o que o sistema pode e não pode computar*
  - *O sistema é capaz de decidir se uma entrada é válida/inexperada/maliciosa & seguramente rejeitá-la?*
  - *Podemos confiar que o sistema nunca fará X, Y, Z?”*
- “Exploração é uma computação inesperada causada de forma estável e provável (probabilisticamente falando) por algumas entradas de dados (criadas/controladas)”

# Dado x Código

- Dado: Todo e qualquer tipo leitura ou escrita na memória principal do computador SEM o propósito de interferir nas operações de computação
- Código: Todo e qualquer tipo leitura ou escrita na memória principal do computador COM o objetivo de interferir nas operações de computação

# Dado x Código

- Dado: Todo e qualquer tipo leitura ou escrita na memória principal do computador SEM o propósito de interferir nas operações de computação
- Código: Todo e qualquer tipo leitura ou escrita na memória principal do computador COM o objetivo de interferir nas operações de computação

CONCORDAM??

# Dado x Código

- Dado: Todo e qualquer tipo leitura ou escrita na memória principal do computador SEM o propósito de interferir nas operações de computação
- Código: Todo e qualquer tipo leitura ou escrita na memória principal do computador COM o objetivo de interferir nas operações de computação

EU NÃO! (pelo menos, não mais!)

# Decidibilidade

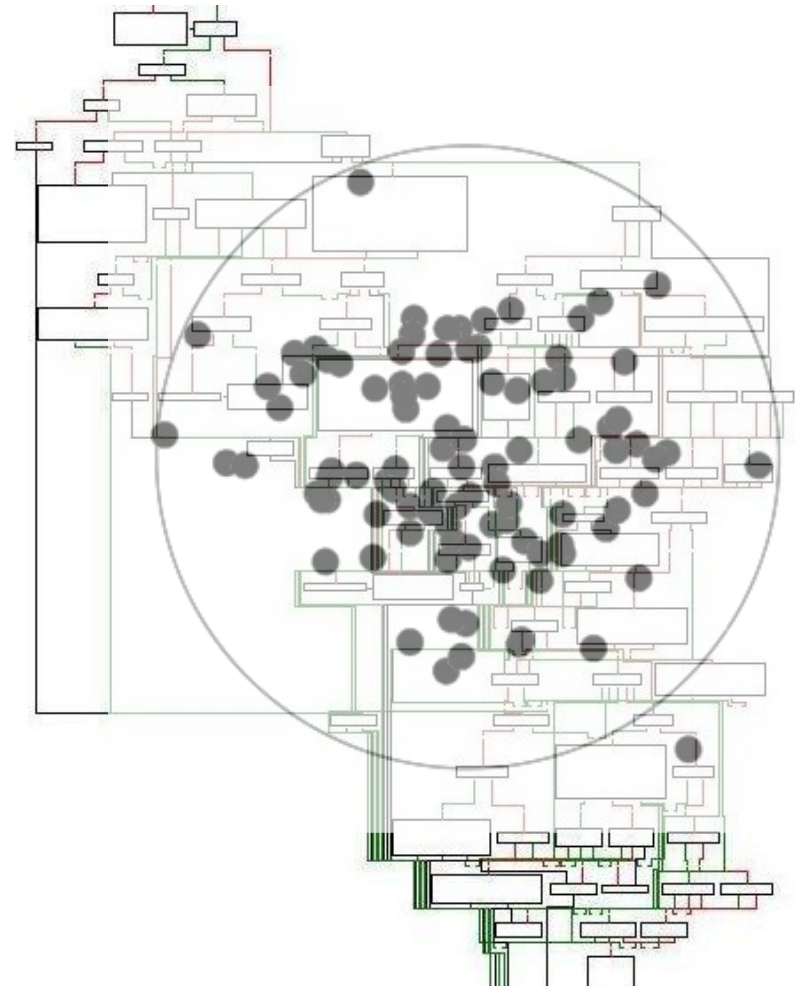
- Computação possui problemas indecidíveis – inclusive sobre reconhecimento de entradas
- Um problema indecidível é aquele em que não existe um algoritmo que resolva para o caso geral

# Exploração de Software

- Acontece quando o código interno de um programa recebe algo que ele não esperava (falha no reconhecimento de entradas)
- Primitivas do programa são expostas
  - Corrupção de memória, fluxos de dados implícitos
  - Fluxos de dados inesperados, etc, etc
- Uma máquina bizarra é criada – “weird machine – Sergey Bratus”
  - Um ambiente de execução programável e mais poderoso do que o intencionado ou esperado

# Software é Complexo

- Validação de entradas distribuída pelo código do programa, misturada com a lógica do mesmo  
- “Shotgun Parser”
- Diversas computações adicionais existentes e disponíveis ao programador da “weird machine”, aka, exploit writer



***“Exploitation is setting up, instantiating, and programming a weird machine” –  
Halvar Flake, Infiltrate 2011***

- Uma parte do programa é sobrecarregada com entradas manipuladas e entra em um estado inesperado e manipulável
- EXPLOIT nada mais é do que um programa para a WM, escrito através das (linguagem limitada pelas) entradas do software original
- Tais entradas manipulam as computações inesperadas que acontecem na WM



# Explorando computações adicionais

## Partes II e III

- Finalmente chegamos ao que interessa...
- Existem diversas computações acontecendo e que podem ser utilizadas na exploração de software
- Ano passado, discuti o caso de computadores com memória criptografada (completamente, ou com chaves diferentes para cada máquina virtual)

# 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?**
  - **What if memory is encrypted?**

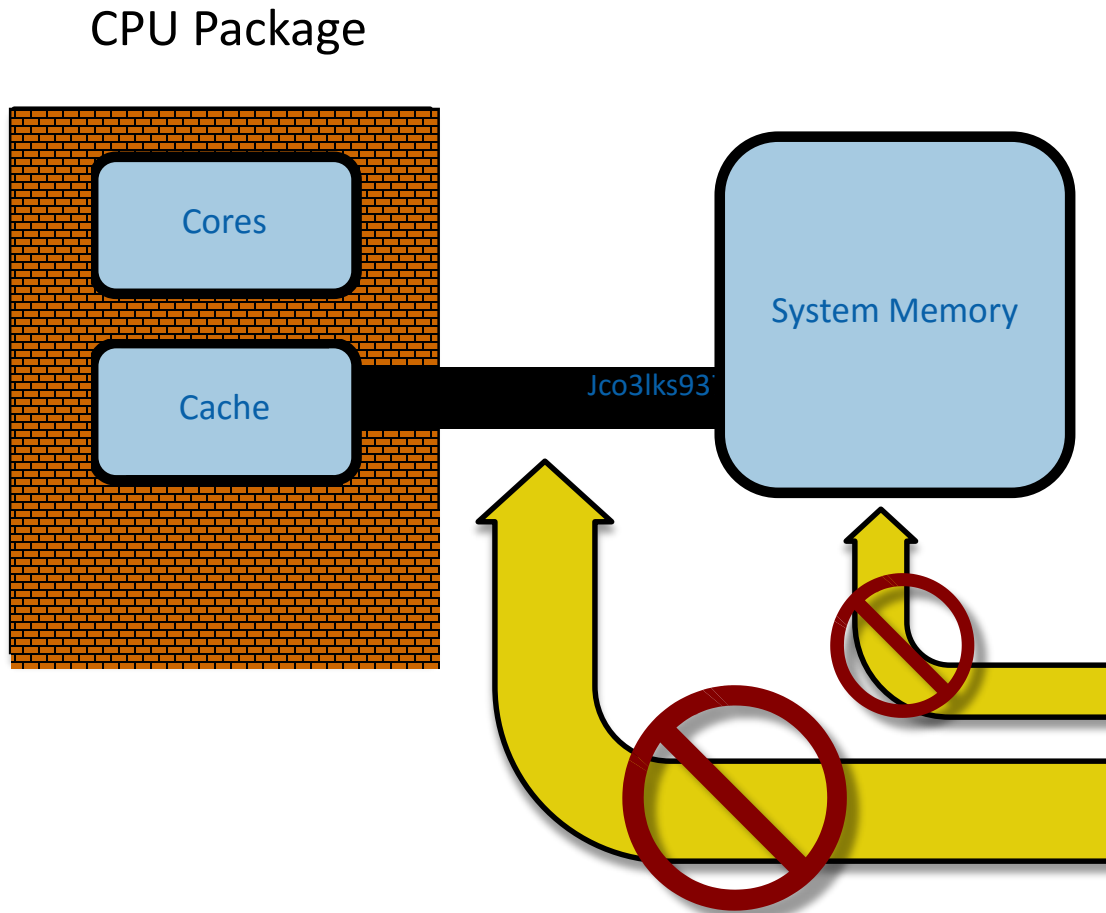
Underlying attack assumption on the threat model:  
The attacker has physical means to modify DRAM

# Transparent memory encryption

- Proposed technologies against active attacks were
  - 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”
  - Seems to be effective for limiting active dynamic attacks
    - Although the physical memory modification **capabilities remain available**
    - Attacker has **limited control** on the result of active attacks

Underlying attack assumption: attacker has physical means to modify DRAM

# 1 image, 1000 words?



1. Security perimeter is the CPU package boundary
2. Data and code unencrypted inside CPU package
3. Data and code outside CPU package is encrypted
4. External memory reads and bus snoops see only encrypted data

# Blinded Random Block Corruption



- Under memory encryption, the attacker has limited capabilities
  - **Blinded Random Block Corruption (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.
  - Expect **randomly corrupted data** to be the result of the decryption
- **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

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

```
global var1...varn
global preauth_flag
global preauth_related
global preauth_enabled
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
global preauth_enabled
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)
  - **Remember that limitation for the attack: conditionals comparing with 0**



# The cloud scenario

## Hypervisor has management interfaces

- VM Introspection capabilities exist for legitimate 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)

# Memory encryption with VM-unique keys

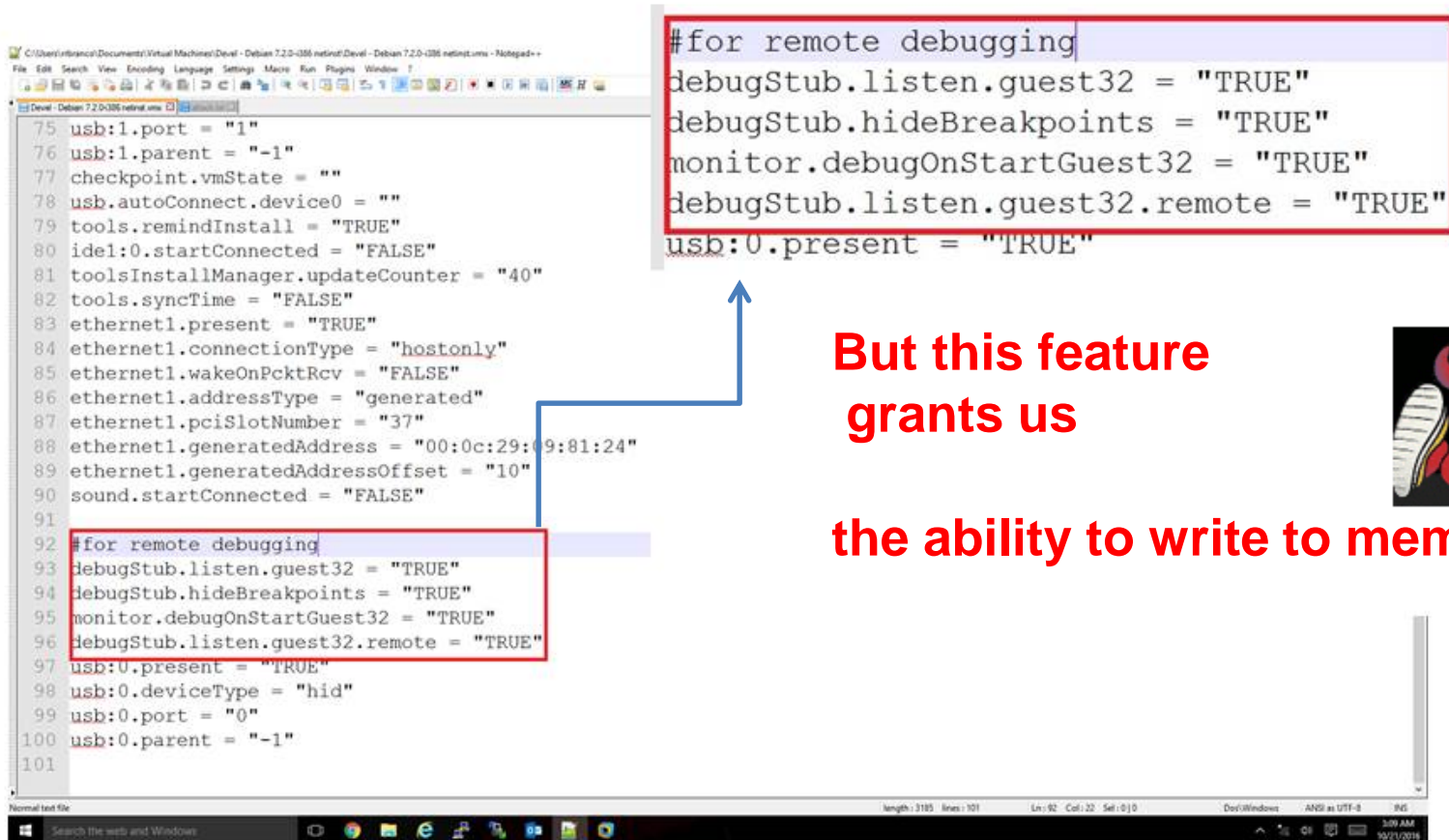
## The threat model

- 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**

## 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



```
75 usb:1.port = "1"
76 usb:1.parent = "-1"
77 checkpoint.vmState = ""
78 usb.autoConnect.device0 = ""
79 tools.remindInstall = "TRUE"
80 ide1:0.startConnected = "FALSE"
81 toolsInstallManager.updateCounter = "40"
82 tools.syncTime = "FALSE"
83 ethernet1.present = "TRUE"
84 ethernet1.connectionType = "hostonly"
85 ethernet1.wakeOnPcktRcv = "FALSE"
86 ethernet1.addressType = "generated"
87 ethernet1.pciSlotNumber = "37"
88 ethernet1.generatedAddress = "00:0c:29:09:81:24"
89 ethernet1.generatedAddressOffset = "10"
90 sound.startConnected = "FALSE"
91
92 #for remote debugging
93 debugStub.listen.guest32 = "TRUE"
94 debugStub.hideBreakpoints = "TRUE"
95 monitor.debugOnStartGuest32 = "TRUE"
96 debugStub.listen.guest32.remote = "TRUE"
97 usb:0.present = "TRUE"
98 usb:0.deviceType = "hid"
99 usb:0.port = "0"
100 usb:0.parent = "-1"
101
```

But this feature  
grants us

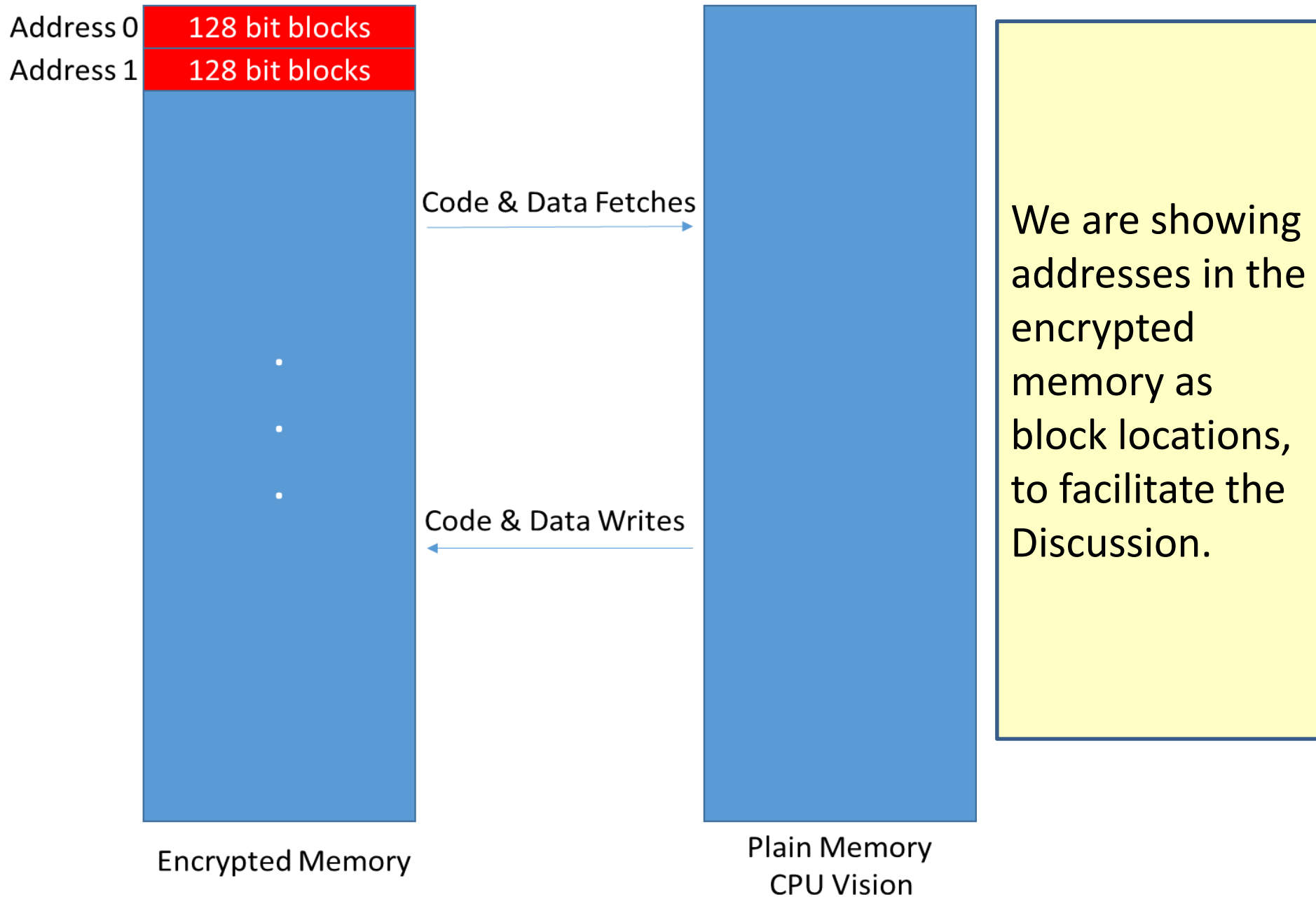
the ability to write to memory



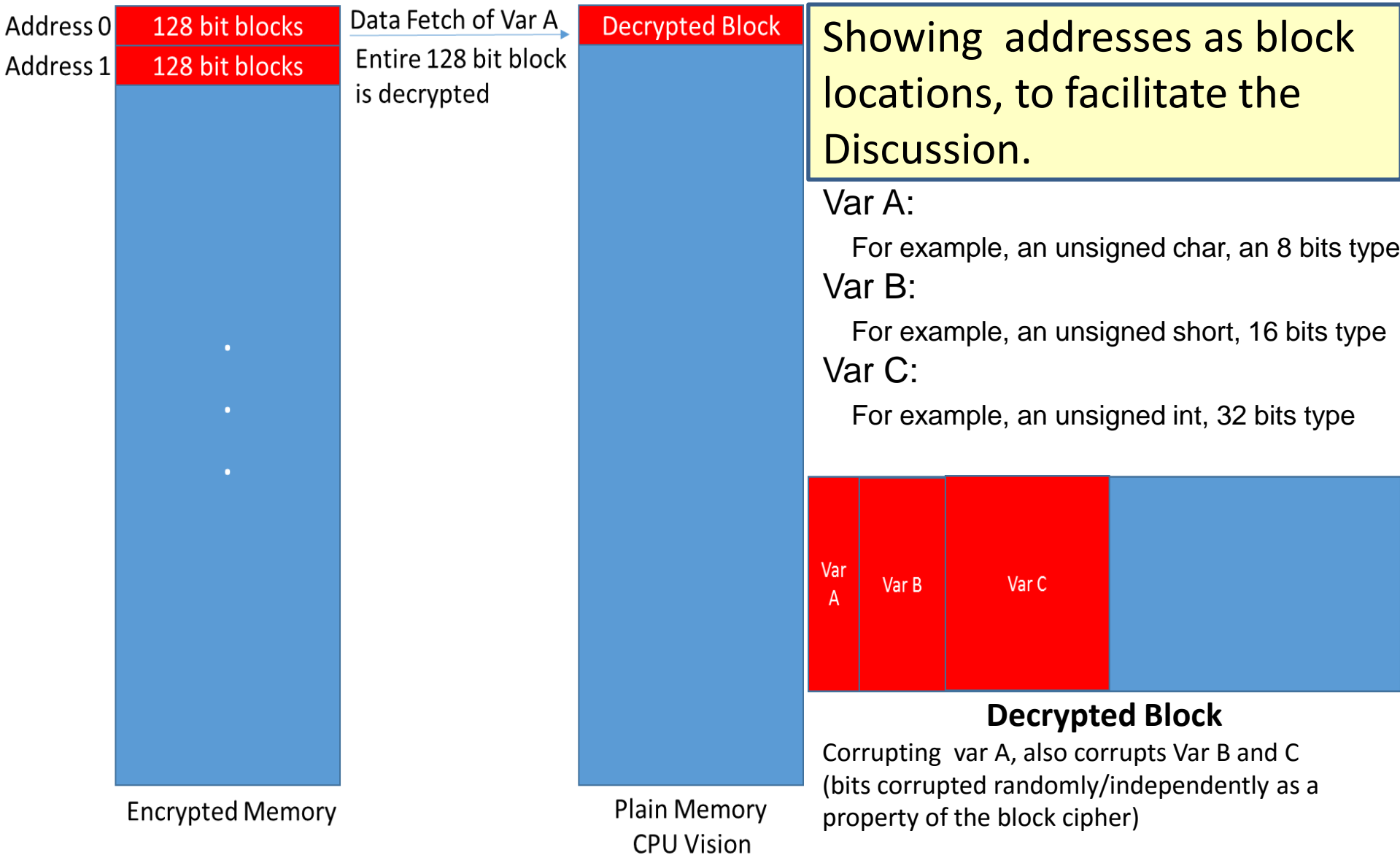
# So, what is new for this year?

- The main limitation of our previous approach was that we needed conditionals comparing with not 0 values that were beneficial to an attacker
  - While such conditionals are frequent, some might argue that they can be easily eliminated by software (automatically even, with compiler changes)
- But we still believed that the attacker premisses are way too powerful to be so limited, and we were right!

# Not only “zero conditional” anymore



# Block corruption independently randomly corrupt each element on the block



# Introducing: Feasible Brute-force-based BRBC

- If we find a way to brute-force a block that has this characteristics:
  - Many different data elements, with different sizes
  - One of those elements being of interest, and small enough to be fully brute-forced (like a 32 bit integer)
    - And for which we are able to tell if we somehow have a value we want
  - In which the other elements, if changed, do not affect our interests as an attacker
  - And for which any value would not affect the system stability (meaning: we can repeat the corruption as many times as we want)
- Then we are able to:
  - Have a fully controlled memory overwrite! (we just need to brute-force the element of interest til it randomly has the value of our interest!)

# Can we make a pie with so many ingredients? \*

- Linux Kernel manages processes using a data structure named `task_struct`
- Such struct has lots of elements necessary to store the process information, such as memory areas, opened files, privileges and so on
- For privileges, it uses a pointer to another data structure, which is the credentials... Having a look at it, we have something quite interesting

\* Homage to a famous quote by *Noir*



# Sounded like impossible?

## A bit on the Linux Kernel...



Following the `task_struct` of a process (to find our target), we see there is a process credentials entry, which is a structure that has many elements, of interest we have:

<code>kuid_t</code>	<code>uid;</code>	<code>/* real UID of the task */</code>
<code>kgid_t</code>	<code>gid;</code>	<code>/* real GID of the task */</code>
<code>kuid_t</code>	<code>suid;</code>	<code>/* saved UID of the task */</code>
<code>kgid_t</code>	<code>sgid;</code>	<code>/* saved GID of the task */</code>
<code>kuid_t</code>	<code>euid;</code>	<code>/* effective UID of the task */</code>
<code>kgid_t</code>	<code>egid;</code>	<code>/* effective GID of the task */</code>
<code>kuid_t</code>	<code>fsuid;</code>	<code>/* UID for VFS ops */</code>
<code>kgid_t</code>	<code>fsgid;</code>	<code>/* GID for VFS ops */</code>



Notice that `kuid_t` and `kgid_t` are typedef's to `__kernel_uid32_t` and `__kernel_gid32_t`, which in turn:

```
typedef unsigned int __kernel_uid32_t;  
typedef unsigned int __kernel_gid32_t;
```

# Is alignment inside the target block an issue?

No, since the adjacent elements do not matter for the attack

Option 1  
*eid* is last  
element  
of a block



**Block 1**



**Block 2**

Affects: gid,  
suid, sgid  
(everything  
before it)

Option 2  
*eid* is  
middle  
element of a  
block



**Block 1**



**Block 2**

Affects: sgid,  
egid, fsuid  
(max 2  
before and  
two after it)

Option 3  
*eid* is first  
element of a  
block



**Block 1**



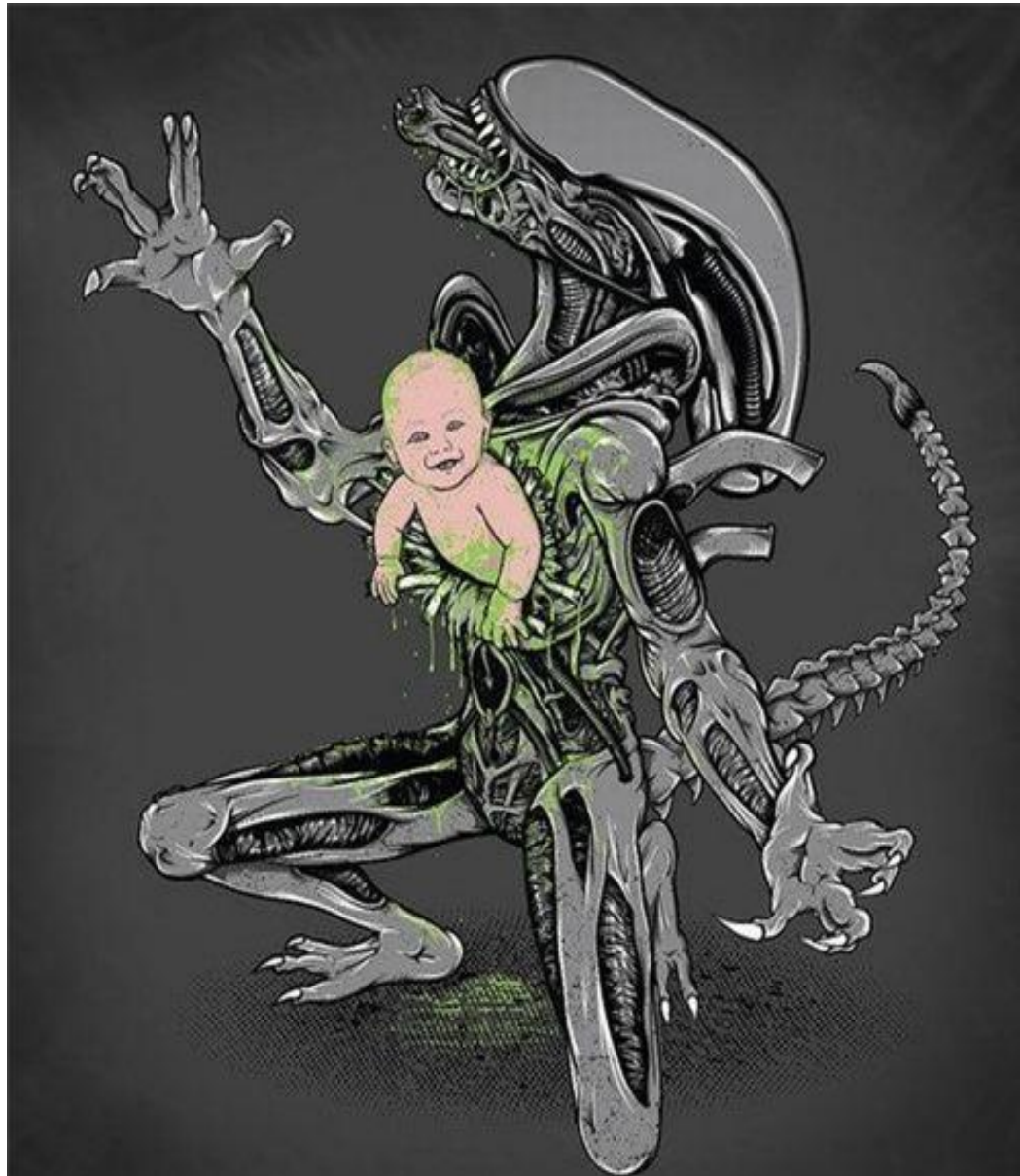
**Block 2**

Affects:  
egid, fsuid,  
fsgid  
(everything  
after it)

# Do we have a winner candidate?

Premisse	Does it satisfy?
Many different data elements, with different sizes	Yes
One of those elements being of interest, and small enough to be fully brute-forced (like a 32 bit integer)	Yes (euid is our target)
And for which we are able to tell if we somehow have a value we want	Yes (our target process has elevated privileges)
In which the other elements, if changed, do not affect our interests as an attacker	Yes (we can change other elements if needed with the privileges)
And for which any value would not affect the system stability (meaning: we can repeat the corruption as many times as we want)	Yes

Who is the founder of this little hacker?



# What about limitations?

- We need to be able to locate such a data structure in memory (we are blind to the memory contents)
  - There are a lot of challenges to being able to do that (and system pressure might affect the ability of doing it)
- For now, our PoC requires a process running on the target (with no privileges)
  - We elevate that process' privilege
  - The requirement for such a process is exactly to avoid the limitation (given we have a process we control, we use such a process to spawn multiple child process and create a predictable memory layout and pattern that we can identify – that is how we locate the correct structure in memory)
  - We are studying other possibilities (like for server-side process that anyhow spawn child processes, and others)

# The end! Is it ??

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

Rodrigo Rubira Branco (@BSDaemon)  
Senior Principal Security Researcher  
Intel Corporation – Security Center of Excellence

**Thank you for your time**