A importância de uma metodologia rigorosa em pesquisa em segurança da informação.

Anderson Nascimento
anderson@allelesecurity.com.br

10º Edição NullByte Security Conference
30 de Novembro de 2024

Agenda

- Sobre
- Introdução
- Metodologia
- Casos
- Conclusão

Disclaimer

- Esta apresentação visa mostrar erros cometidos por mim e pessoas próximas com o objetivo de ajudar outros profissionais a entender, e se possível, evitar cometer os mesmos erros. Erros cometidos durante o processo de aprendizagem são normais e fazem parte do processo de evolução.
- Nesta apresentação, o autor em nenhum momento imagina, acredita ou pressupõe que as pessoas envolvidas são incapazes de realizar as tarefas mencionadas ou que detenham capacidade inferior a ele ou a outrem.
- Esta apresentação representa apenas a opinião do autor, podendo ter erros, vieses e incoerências.

Sobre

- Anderson Nascimento
- Pesquisador em segurança da informação, focado em Android e *kernel* do Linux
  - Mais de 10 anos de experiência com descoberta e exploração de vulnerabilidades.
- Anteriormente, trabalhou como *pentester* e administrador de sistemas
  - Como *pentester*, participei de projetos nacionais e internacionais. Ministrei treinamentos para clientes privados e governamentais.
  - Como administrador de sistemas, trabalhei no gerenciamento de servidores de um provedor de *internet* de uma cidade.
- Fundador e principal pesquisador na Allele Security Intelligence
  - Única empresa focada em descoberta e exploração de vulnerabilidades de baixo nível do Brasil.
  - Ministrei treinamentos para dezenas de profissionais.

Introdução

- Realizo pesquisas em segurança da informação (*vulnerability research*) há mais de dez anos. Por esse motivo, algumas pessoas têm me procurado para ajudar nos estudos de pesquisa e exploração de vulnerabilidades. Além disso, gerencio um time de pesquisadores na Allele Security Intelligence.
- Em minha experiência, um dos principais motivos que dificultam o desenvolvimento na área de pesquisa (descoberta e exploração de vulnerabilidades) é a ausência de uma metodologia rigorosa.
- Sendo assim, o objetivo desta apresentação é mostrar alguns casos em que a metodologia foi um fator importante e os principais erros cometidos, por mim e pelas pessoas que tentei ajudar em minha carreira.

Primeiro, o que é uma metodologia?

Metodologia

“A metodologia é o estudo dos métodos. Isto é, o estudo dos caminhos para se chegar a um determinado fim.

Com o objetivo de analisar as características dos vários métodos indispensáveis tais como: avaliar capacidades, limitações e criticar os pressupostos quanto sua utilização.”

https://pt.wikipedia.org/wiki/Metodologia

Segundo, o que é uma metodologia rigorosa?

Metodologia rigorosa

- Métodos e técnicas aplicadas no estudo de um objeto evitando ao máximo efeitos adversos, errôneos, falsos-negativos ou falsos-positivos, erros lógicos, cognitivos e vieses.
- Conclusões e afirmações baseadas em dados e evidências e não em opiniões ou achismos.
- Redução de efeitos adversos e entendimento dos *confounders*[1].
- Repetições suficientes para garantir que o resultado é válido.
- Experimentos realizados em diversas condições e ambientes.

[1] - https://en.wikipedia.org/wiki/Confounding

O que não é uma metodologia rigorosa?

- Deduzir sem evidências claras
  - Achismos e opiniões sem base sólida. Eu acho que o problema é X, mas qual a base para isso? Quais as evidências?
- Não encontrar a causa raiz. Esgotou as opções, mas não conseguiu entender claramente, então acha que é X.
- Garantir ao máximo que o teste (hipótese) que está realizando (testando) realmente testa (valida) o que está querendo testar (validar).
- Hipótese vaga, propósito não claro
  - É possível testar o que quero testar? Como vou saber se o teste foi bem sucedido?
  - Exemplo:
    - Acho que tem uma vulnerabilidade no subsistema de memória do *kernel* do Linux. Qual componente? Qual impacto? Qual o modelo de ameaça?

Passos de uma metodologia rigorosa

- Tentar invalidar o resultado
  - Será que o resultado realmente prova a hipótese?
- Repetir experimento diversas vezes
- Realizar o experimento em ambientes e condições diferentes
- Reprodutibilidade
  - Outros conseguem reproduzir meus experimentos e obter o mesmo resultado?
- Com uma metodologia rigorosa, o profissional consegue entender melhor o funcionamento do sistema e assim, identificar corretamente os erros sendo cometidos, além de aprender de forma adequada.

Hipótese

"Uma hipótese, suposição ou especulação é uma formulação provisória, com intenções de ser posteriormente demonstrada ou verificada, constituindo uma suposição admissível.

É a evolução da intuição à teorização e da teoria que levará à prática, a testar as hipóteses firmadas pelo raciocínio dedutivo implícito à teorização, com frequência, e por motivos vários, que segue por vias aparentemente obscuras."[1]

[1] - https://pt.wikipedia.org/wiki/hipótese

Documentação

- Consultar documentação para entender como realmente funciona e o comportamento esperado.
  - Ler manpages
    - Chamadas de sistemas, subsistemas do *kernel* do Linux.
  - Ler manuais
    - Processadores e arquiteturas.
  - Ler código-fonte
    - *Kernels*, aplicações e afins.

Navalha de Ockham

- Grosso modo, o princípio postula que de múltiplas explicações adequadas e possíveis para o mesmo conjunto de fatos, deve-se optar pela mais simples daquelas. Por “simples” entende-se aquela que contiver o menor número possível de variáveis e hipóteses com relações lógicas entre si, das quais o fato a ser explicado segue logicamente.
- Em termos simplificados, a Navalha de Ockham pode ser descrita da seguinte forma:
  - Em igualdade de condições, a explicação mais simples é geralmente a mais provável.

Força bruta

- Funciona em alguns casos e é recomendado utilizar quando possível, mas pode impedir o correto aprendizado.
- O profissional poderá ficar viciado se toda vez que tiver um problema, resolvê-lo através de força bruta, sem compreender a causa raiz e os fatores envolvidos. Portanto, ele não empenhará esforços para obter uma melhor compreensão da questão/problema.
- O profissional também poderá ficar mais sensível a fatores psicológicos quando não conseguir resolver através de força bruta e desistir.

Problemas metodológicos em pesquisa de vulnerabilidade

- *Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability, Kyle Zeng et al*.
  - Em estudo objetivando entender melhor as técnicas de escrita de *exploit* para o *kernel* do Linux. Foi descoberto que *experts* em escrita de *exploits* têm opiniões incorretas sobre técnicas de estabilização de *exploits*.
    - https://www.usenix.org/system/files/sec22-zeng.pdf
- Tentativa de descobrir o que hoje conhecemos como Meltdown
  - Blog post escrito por Anders Fogh com a hipótese correta, porém obteve um resultado negativo. Em contato direto com ele, ele menciona não saber o motivo do resultado negativo. Após o blog post, ele obteve um resultado positivo. Perguntei se ele descreveria o resultado negativo como uma falha de metodologia e ele respondeu que não sabia dizer.
    - https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/

“The dynamic of the Linux kernel heap layout significantly impacts the reliability of kernel heap exploits, making exploitability assessment challenging. Though techniques have been proposed to stabilize exploits in the past, little scientific research has been conducted to evaluate their effectiveness and explore their working conditions. In this paper, we present a systematic study of the kernel heap exploit reliability problem. We first interview kernel security experts, gathering commonly adopted exploitation stabilization techniques and expert opinions about these techniques. We then evaluate these stabilization techniques on 17 real-world kernel heap exploits. **The results indicate that many kernel security experts have incorrect opinions on exploitation stabilization techniques.** To help the security community better understand exploitation stabilization, we inspect our experiment results and design a generic kernel heap exploit model. We use the proposed exploit model to interpret the exploitation unreliability issue and analyze why stabilization techniques succeed or fail. We also leverage the model to propose a new exploitation technique. Our experiment indicates that the new stabilization technique improves Linux kernel exploit reliability by 14.87% on average. Combining this newly proposed technique with existing stabilization approaches produces a composite stabilization method that achieves a 135.53% exploitation reliability improvement on average, outperforming exploit stabilization by professional security researchers by 67.86%.”

https://www.usenix.org/system/files/sec22-zeng.pdf

Problemas metodológicos em pesquisa de vulnerabilidade

- *From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel*
  - Artigo de 2015 sobre uma técnica de exploração de vulnerabilidades no *kernel* do Linux.
    - https://dl.acm.org/doi/10.1145/2810103.2813637
  - Tentei reproduzir essa técnica em 2015/2016, mas não obtive êxito. Era muito inexperiente metodologicamente para poder reproduzir de forma efetiva.
  - Escrevi um *exploit* este ano que utiliza técnicas modernas e desconhecidas, mais avançadas do que a técnica apresentada no artigo. Escrevi o *exploit* rápido e de forma confiável.
    - Isso mostra o quanto aprendi desde 2015/2016.

Metodologia rigorosa/robusta

- Entender bem o que está sendo testado
- O ambiente está adequado? Consigo reproduzir em outro ambiente?
- Visibilidade
  - O que pode influenciar meus resultados?
- A detecção (monitoramento) é efetiva?
  - Eu consigo fielmente obter o resultado?
  - O resultado é realmente válido ou é um ruído? Tive sorte? Muito comum em análises temporais.
- Reprodutibilidade
  - Consigo ter consistência entre os resultados?
- Revalidar cuidadosamente os resultados. Recomendado ser feita por um terceiro, se possível.

Exemplos de erros metodológicos

Caso 01

Uso incorreto de API da linguagem GO

Caso 01 - Uso incorreto de API da linguagem GO

- Desenvolvemos uma aplicação (x) que coletava arquivos de uma aplicação ***A***, através de sua API e enviava para uma aplicação de terceiros, ***B***, também através de sua API.
- Os arquivos eram no formato XML. Em determinado momento, percebemos que alguns arquivos não estavam sendo recebidos. Na aplicação ***B***, não era localizado determinados arquivos.
- Um profissional dedicou tempo para identificar o que estava acontecendo. Depois de muito tempo dedicado, mais de um mês, foi reportado como um problema da aplicação ***B***. Ela tinha sido atualizada entre testes em que tudo tinha funcionado como esperado.
- Não tenho claras lembranças do que aconteceu depois daí, mas lembro bem que chegamos a entrar em contato com a empresa desenvolvedora e que o problema ainda continuava.

Caso 01 - Uso incorreto de API da linguagem GO

- A aplicação, escrita em linguagem Go, coletava dados de ***A*** através da API, armazenava no disco e utilizando a API da aplicação ***B***, importava os dados.

A <-> aplicação (x) -> B

Caso 01 - Uso incorreto de API da linguagem GO

- Hipótese do profissional
  - O problema foi notado em uma versão mais nova da aplicação ***B***. Durante testes em versões anteriores da aplicação, todos os dados foram importados com sucesso. Sendo assim, após exaurir as opções, foi sugerido que o problema era na API aplicação ***B***.
- Depois que percebi que o problema não seria resolvido e que já tinha tempo que estava em aberto, decidi começar a analisá-lo.
- Resultado
  - A aplicação que desenvolvemos continha um erro que escrevia o conteúdo do arquivo no disco usando a API da linguagem que interpretava o conteúdo do XML como *format string*. Quando o XML continha caracteres com porcentagem (%), o arquivo XML escrito em disco era corrompido e rejeitado pela aplicação B.

Caso 01 - Uso incorreto de API da linguagem GO

```
558 func saveOnDisk(dir string, StrfileName string, data string) error {
559         fileName := fmt.Sprintf(dir + StrfileName)
560
561         file, err := os.Create(fileName)
562
563         if err != nil {
564                 return errors.Wrap(err, "Save on disk falhou")
565         }
566
567         defer file.Close()
568         //[COMMENT] essa funcao ferra os caracteres especiais no texto ao salvar o arquivo
569         // fmt.Fprintf(file, data)
570
571         file.WriteString(data)
572
573         return nil
574
575 }
```

Caso 01 - Uso incorreto de API da linguagem GO

- Resumo
  - O problema acontecia somente com alguns dados. O problema já existia antes da aplicação ***B*** ter sido atualizada, porém, devido aos dados utilizados nos testes anteriores não conter o caractere porcentagem (%), os testes não falharam.
  - Profissional passou meses tentando entender o problema, não chegou na causa raiz, culpou o componente errado (aplicação ***B***) e não solucionou o problema.

Caso 01 - Uso incorreto de API da linguagem GO

- Lições
  - Pensar nos efeitos colaterais (efeitos de ordem superior) ao escrever código e aplicações.
  - Entender a causa raiz do problema, se possível. Se não for possível, tentar obter um alto grau de confiança antes de apontar culpados.
  - Analisar cuidadosamente todos os componentes envolvidos. Repetir testes diversas vezes em condições e ambientes diferentes, mesmo tendo resultados positivos/negativos anteriormente.
  - Entenda claramente seus objetivos.

Caso 02

Registradores 64 bits em operações 32 bits na arquitetura AMD64

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

- Profissional queria validar um recurso específico da arquitetura AMD64 (64 bits).
- Na arquitetura AMD64, operações de 32 bits em registradores 64 bits resultam nos 32 bits superiores zerados (*zero-extended*).
  - `mov $0x4433221144332211,%rax`
  - `mov $0xffffffff,%eax`
- Após as operações *assembly* acima, o registrador RAX deverá ter o resultado `0xffffffff` e não `0x4433221fffffffff`.

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

- Profissional me abordou questionando se (Intel) mudou a arquitetura, porque este recurso não estava funcionando em seu código como ele esperava.
- Suposição do profissional
  - Após consultar um amigo, era problema de *cache*. O *cache* poderia estar influenciando o código. Não ficou claro qual *cache*. Processador? Compilador?
- Não existe um "*cache*" do compilador, mas nesses casos, não sabemos o que o profissional estava pensando e precisamos deduzir e supor o que ele quer dizer. Talvez o profissional tivesse um entendimento incorreto dos *caches* envolvidos em um computador.
- Não tive tempo de imediato para analisar, porém depois de alguns dias, perguntei se já tinha resolvido.

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

- Respondi de forma clara e direta.
  - O recurso sempre (desde quando foi introduzido) aconteceu. É difícil isso não acontecer onde deveria acontecer. Se não está acontecendo, é porque realmente (não atende às condições) não deveria ocorrer.
  - Profissional respondeu que ainda suspeita que era o *cache* porque quando mudava o registrador, funcionava. Então, a suposição era que o `printf()` estava consultando o *cache* ao invés de ler o registrador.
  - Eu já imaginava onde estava o problema (no código e não no recurso do processador) e perguntei se não queria reescrever todo o código em *assembly*, que seria muito mais simples e direto.
  - Profissional mencionou que escrever código em *assembly* e imprimir na tela para mostrar o resultado ficaria mais complicado do que gostaria.

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

- Eu não me aprofundei no código, não tinha motivos para isso. Meu objetivo era saber o objetivo do profissional com o código e ajudá-lo a alcançá-lo.
  - Apenas confirmar que o recurso funciona? Disponibilizar o código para outros verem o recurso funcionando?
- Entendi, através de conversas com o profissional, que ele tinha várias suposições erradas. O recurso é muito mais simples de ver e entender usando apenas *assembly*. Não precisa usar *syscalls* para escrever o resultado na tela. Pode apenas usar o GDB para inspecionar os registradores e mostrar os resultados.

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

- Após entender que independentemente do problema no código, a forma utilizada não era a melhor para o objetivo do profissional. Então, recomendei usar *assembly* e reescrever o teste e usar o GDB para mostrar o recurso em ação. Mas de qualquer forma, fui dar uma olhada no código.
- O código continha vários erros. Não erros no código, mas erros na suposição do profissional.
  - Se você escrever X, o código vai fazer X. Ele não está errado em si, mas se você espera Y e escreve X, o erro está em seu entendimento, modelagem e suposição.
- O recurso não funcionava como esperado porque o código escrevia o valor na memória e não no registrador.
  - O recurso funciona com registradores e não dados em memória.
  - O código estava escrevendo em memória devido ao uso de *assembly inlining*.

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

- O profissional acreditava que usar apenas `asm("REGNAME")` na declaração de uma variável local em C faria com que a variável utilizasse o registrador `REGNAME` para armazenar o conteúdo da variável. Suposição incorreta. Para isso ocorrer, precisa também inserir a *keyword* `register`.
  - `register int *foo asm ("r12");`
- Após eu mencionar que nunca tinha visto essa *keyword* ser usada para esse propósito e acreditar que não funcionava como o profissional esperava, o profissional mencionou que testou e funcionou. Ele validou e aparentemente confirmou que funcionava.
- Claramente a metodologia utilizada para validar o uso da *keyword* `asm()` para esse propósito não foi rigorosa, pois não é assim que funciona.

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

- A *keyword* `asm()` até funciona como o profissional esperava, porém precisa usar em conjunto com a *keyword* register.
- Da forma como o profissional utilizou, REGNAME serve apenas como uma *label*/nome da variável no código gerado.

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

6.50.4 Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C function or variable by writing the `asm` (or `__asm__`) keyword after the declarator. It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols, or reference registers.

Assembler names for data

This sample shows how to specify the assembler name for data:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable `foo` in the assembler code should be '`myfoo`' rather than the usual '`_foo`'.

On systems where an underscore is normally prepended to the name of a C variable, this feature allows you to define names for the linker that do not start with an underscore.

GCC does not support using this feature with a non-static local variable since such variables do not have assembler names. If you are trying to put the variable in a particular register, see Variables in Specified Registers.

https://gcc.gnu.org/onlinedocs/gcc/Asm-Labels.html

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

6.50.5.2 Specifying Registers for Local Variables

You can define a local register variable and associate it with a specified register like this:

```
register int *foo asm ("r12");
```

Here `r12` is the name of the register that should be used. Note that this is the same syntax used for defining global register variables, but for a local variable the declaration appears within a function. The `register` keyword is required, and cannot be combined with `static`. The register name must be a valid register name for the target platform.

https://gcc.gnu.org/onlinedocs/gcc/Local-Register-Variables.html

Caso 02 - Registradores 64 bits em operações 32 bits na arquitetura AMD64

- Resumo
  - Simplificar ao máximo os testes. Evitar códigos e artefatos desnecessário. Um computador já é complexo o suficiente, não precisamos dificultar ainda mais.
  - Não confiar nem em nós mesmos. Cometemos muitas falhas e é normal.
  - Navalha de Ockham
    - O que é mais provável? Os códigos e testes estarem errados ou as fabricantes de processadores terem modificado um recurso válido e essencial desde sempre?
- Lições
  - Utilizar metodologia rigorosa para validar as hipóteses, suposições e conhecimento.
  - Consultar documentação oficial quando existir.
  - Criar hipóteses fundamentadas.
    - Problema de *cache*? Qual *cache*? Como o *cache* estaria afetando o comportamento esperado?

Caso 03

Entrada obsoleta na TLB (TLB *stale entry*)

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- *Translation Lookaside Buffer* (TLB) *stale entry*
  - Uma das classes de vulnerabilidades mais interessantes, em minha opinião.
  - Ela permite a um código acessar a memória de forma que naturalmente não seria permitida.
  - O problema acontece quando ocorre uma dessincronização entre a TLB e as tabelas de página (*page tables*).

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- *Translation Lookaside Buffer* (TLB) *stale entry*
  - A tabelas de página contém informações de tradução de endereços virtuais para endereços físicos de um processo.
    - Processo A, o endereço virtual X aponta para endereço físico Y.
  - Porém, na TLB, ainda consta uma entrada do endereço virtual X apontando para endereço físico Z. Dessa forma, o processo A consegue acesso de leitura, escrita ou execução de forma que não seria permitido naturalmente para aplicação.
    - A página física Z pode estar sendo utilizada por um outro processo ou contendo objetos importantes do *kernel*.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
Subject: Linux kernel: Security sensitive bug in the i915 kernel driver (CVE-2022-0330)


[This is a public disclosure of an issue reported 7 days ago to
linux-distros at openwall. CVE-2022-0330 has been assigned to the
issue since.]

Hi all,

A missing GPU TLB flush has been discovered in the i915 kernel driver
which could be exploited by malicious userspace and can manifest in two
flavours, depending on whether the GPU is running behind an active IOMMU
(address translation) or not:

1. IOMMU with address translation - malicious userspace can trigger DMAR
read/write faults getting logged to the kernel log.

2. Without an active IOMMU malicious userspace can gain access (from the
code executing on the GPU) to random memory pages.

Second case is therefore the serious one.

It is currently not known whether specific memory could be targeted, but
random memory corruption or data leaks are a known possibility.

Underlying reason for the access to memory not owned is a missing TLB
flush upon releasing memory which used to back a GPU buffer object back
to the system.
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
Flawed assumption was that flushing the TLB at the start of every
userspace GPU execution is sufficient, given the programming model where
userspace is expected to declare which graphics virtual memory address
ranges it will be accessing at the start of every execution. However
what was not considered is that userspace can legitimately (it is
allowed in uapi) _not_ declare those accesses.

This allows userspace to continue GPU access to memory, while the kernel
driver (i915) is unaware of it being in use, and therefore is allowed to
release the backing store back to the system. Should the system then
give out those pages back for a different use, the exploit situation can
arise.

Return of the pages back to the system can either be specifically
engineered by the malicious software, or can happen innocently via
system memory pressure.

All Intel integrated and discrete GPUs starting from Gen8 (Broadwell)
are affected.

Fix has already been developed and consists of explicitly flushing the
TLBs before releasing memory back to the system for any GPU buffer
objects which were in use from the GPU.

Note that this will have a varying performance impact depending on the
specific GPU, GPU workload and overall system workload.

Fix for the issue has been provided to the Linus distributions and Linux
kernel maintainers and is expected to be merged to top of the tree and
stable and LTS releases shortly. Fix carries the title of "drm/i915:
Flush TLBs before releasing backing store".

Kind regards,
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
Subject: Security sensitive bug in the i915 kernel driver (CVE-2022-4139)

Hi all,

[This is a public disclosure of an issue reported 7 days ago to
linux-distros@...openwall.org. CVE-2022-4139 has been assigned to the
issue since.]

Incorrect GPU TLB flush code has been discovered in i915 kernel driver.
In some cases (Gen12 hardware with specific types of engine) the
engine's TLB is not flushed at all.
Depending on whether the GPU is running behind an active IOMMU there are
two possible scenarios which can happen, due to stale TLB mapping:
1. Without IOMMU - GPU can still access physical memory which could be
already assigned by OS to different process.
2. With IOMMU - GPU can access any memory, if the malicious process is
able to create/reuse necessary IOMMU mappings.

It is currently not known if specific memory could be targeted, but
random memory corruption or data leaks are a known possibility.

All Intel integrated and discrete GPUs Gen12 are affected, including
Tiger Lake, Rocket Lake, Alder Lake, DG1, Raptor Lake, DG2, Arctic
Sound, Meteor Lake.
Fix has already been developed and consists of fixing the method of
writing to specific registers.
I am attaching a set of back-ported patches which implement the fix for
all affected stable branches (all since 5.4).
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Experimento
  - Realizar testes, aprender e reproduzir vulnerabilidades desta classe.
  - Porém, cometi diversos erros no processo. Os principais erros foram erros metodológicos.
- Erros metodológicos
  - Primeiro erro
    - Assumir que eu entendia bem sobre paginação da arquitetura AMD64
  - Segundo erro
    - Utilizar o *debugger* de *kernel* (GDB) para alterar as entradas das tabelas de páginas.
  - Terceiro erro
    - Utilizar printf() para acessar o endereço de memória.
  - Quarto erro
    - O ambiente de trabalho também não foi o ideal.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Primeiro erro
  - Assumir que após a TLB ser limpa, um *page fault* ocorrerá ao acessar endereço virtual.
    - Não lembro o contexto exato, mas as mensagens mostram que antes de realizar o teste direto de TLB *stale entry*, eu verifiquei esse comportamento específico.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

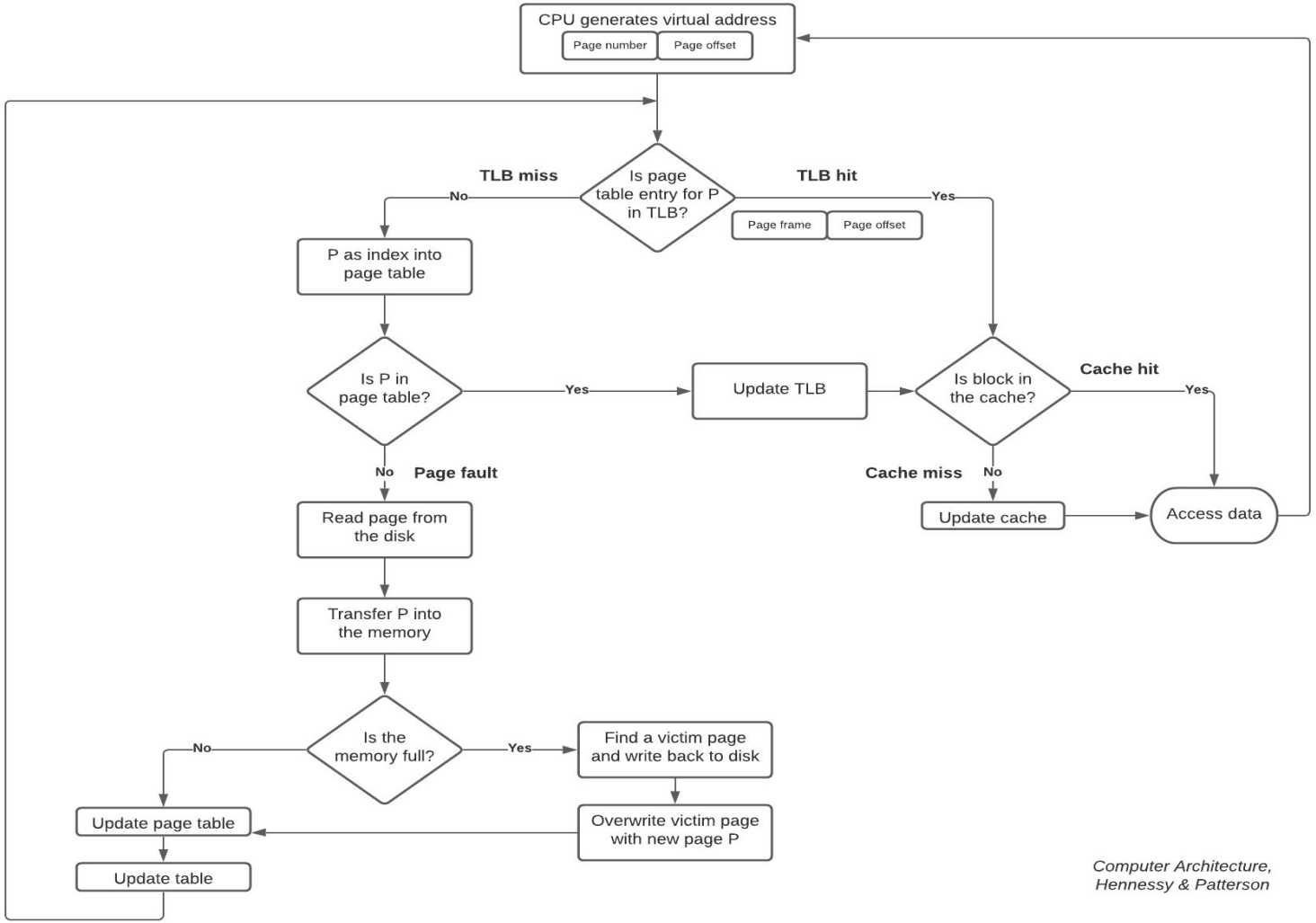
- Primeiro erro
  - Hipótese
    - Um *page fault* deve ocorrer após realizar o *flush* de um endereço na TLB.
    - Comportamento esperado:
      - *Page fault*.
    - Comportamento obtido:
      - O acesso procede normalmente, sem um *page fault*.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Primeiro erro
  - Hipótese
    - Por que?
      - Porque é assim que funciona. O meu conhecimento e suposição que estavam incorretos.
    - Teste realizado:
      - Acessar o endereço normalmente (primeiro acesso, um *fault* ocorrerá inicialmente e após o acesso ser realizado, a TLB será populada).
      - Realizar o *flush* do endereço na TLB
      - Re-acessar o endereço.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Primeiro erro
  - Assumir que após a TLB ser limpa, um *page fault* ocorrerá.
    - Em retrospecto, pode parecer simples, mas eu não entendia bem como funcionava e passei horas (se não dias) tentando obter um resultado de acordo com o que eu esperava. Um computador é bastante complexo. Existem diversos componentes e cenários envolvidos para cada recurso presente e é realmente complicado entender cada detalhe.
    - Precisei atualizar meu entendimento, eu nunca iria concluir os testes com sucesso porque minha suposição estava errada.
    - Após esgotar as opções, entrei em contato com uma pessoa pelo Twitter. Antes da pessoa responder, de alguma forma, eu entendi que minha suposição de como um recurso do processador funcionava estava errada.
    - As respostas que obtive pelo Twitter me ajudaram bastante. A pessoa foi bastante solícita e esclareceu algumas dúvidas que eu tinha.



Computer Architecture, Hennessy & Patterson

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Segundo erro
  - Utilizar *debugger* para modificar as tabelas de páginas.
- Ambiente
  - Escrevi um módulo do *kernel* que altera a PTE (*Page Table Entry*) de um endereço de um processo.
    - Endereço virtual x1 -> Endereço físico p1
    - Endereço virtual x2 -> Endereço físico p2

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Segundo erro
  - Utilizar *debugger* para modificar as tabelas de páginas.
- Ambiente
  - Após o processo alvo executar uma função do módulo do *kernel* via ioctl().
    - Endereço virtual x2 -> Endereço físico p1
  - As tabelas na memória serão sobrescritas, porém como não há *flush* da TLB, a informação anterior, x2 -> p2, ainda deverá constar na TLB e o resultado do acesso não deverá ser alterado.
    - Ler x1 retorna conteúdo de p1
    - Ler x2 retorna conteúdo de p2

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Segundo erro
  - Utilizar *debugger* para modificar as tabelas de páginas.
- Ambiente
  - Isso evita o problema do tempo ao modificar usando o GDB.
    - Fornece tempo suficiente para ocorrer um *flush* da TLB em paralelo. Devido a esse motivo e os outros erros, o código não consegue acessar uma entrada obsoleta na TLB e não consigo completar meu experimento.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Segundo erro
  - Utilizar *debugger* para modificar as tabelas de páginas.
- Ambiente
  - Utilizar o debugger de *kernel* (GDB) para alterar as entradas das tabelas de páginas.
    - No cenário que foi utilizado, uma máquina virtual rodando no VMware, o *debugger* de *kernel* interrompe a execução da máquina virtual. Porém, isso não garante que a TLB não será limpa.
    - Não há uma TLB reservada para a máquina virtual. A TLB será compartilhada entre todos os códigos em execução na máquina, incluindo as máquinas virtuais e os processos em execução no *host*. Sendo assim, o tempo que leva com a máquina interrompida e a modificação da *page table* no *debugger* permite que a TLB seja limpa, inclusive pelo sistema operacional do *host*, gerando um resultado negativo ao meu experimento.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Terceiro erro
  - Usar `printf()`
- Tempo de acesso
  - Em muitos experimentos o tempo é um fator crucial.
  - O código estava usando `printf()`.
    - `printf()` é uma função da libc. Muito código será executado até que o endereço de memória desejado seja lido.
    - A função da libc `printf()` leva muito tempo para realizar a leitura da memória, fornecendo tempo suficiente para uma limpeza (*flush*) da TLB ocorrer em paralelo.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Terceiro erro
  - Usar `printf()`
- Tempo de acesso
  - Em muitos experimentos o tempo é um fator crucial.
  - O código estava usando `printf()`.
    - Por mais que o código ou o *core* (processador) não realize o *flush* diretamente, um outro *core* pode solicitar o *flush* em todos os *cores* através de um IPI (*Inter-Processor Interrupt*). Este mecanismo é conhecido como TLB *shootdown*.
    - Substituir `printf()` por uma leitura usando *assembly inlining*.
      - Por mais que essa substituição melhore consideravelmente o experimento, ainda existia um *race condition* que fazia com que o código funcionasse de forma não confiável. Era necessário executar o código mais de uma vez para poder ter o comportamento esperado. O código não foi otimizado para ser *reliable*.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Quarto erro
  - Ambiente de trabalho não ideal.
- Ambiente
  - Utilizei máquinas virtuais com o VMware, além de ter outras máquinas virtuais em execução e um uso normal do computador (emails, redes sociais, navegação, etc.).
  - O ambiente ideal deve realizar os experimentos na máquina *host*, sem virtualização e com uso apenas para o propósito dos testes.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Quarto erro
  - Ambiente de trabalho não ideal.
- Ambiente
  - Um teste específico, remover o bit P (*Present*) da tabela de página (PTE) e utilizar a entrada obsoleta na TLB, não funcionou como esperado. O experimento funcionou após reiniciar a máquina e realizar o experimento com apenas a máquina virtual desejada em execução. Remoção de ruído.
  - Antes de conseguir fazer funcionar qualquer teste, não sabia como o VMware funcionava e precisei testar em outros ambientes, incluindo QEMU. Isso levou bastante tempo, pois o QEMU não estava funcionando.

Caso 03

Entrada obsoleta na TLB
(TLB *stale entry*)

Alterando a PTE via *debugger*.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
$ ./code01
PID: 2214
address mapped: 0x414243000

writing to addresses

reading and writing to the addresses
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
crash> vtop -c 2214 0x414243000
VIRTUAL     PHYSICAL
414243000   101f4000

   PGD: f2e0000 => 8608067
   PUD: 8608080 => 7851067
   PMD: 7851508 => ded0067
   PTE: ded0218 => 101f4867
  PAGE: 101f4000

  PTE      PHYSICAL  FLAGS
101f4867  101f4000  (PRESENT|RW|USER|ACCESSED|DIRTY)

      VMA           START       END     FLAGS FILE
ffff88800acd6640  414243000  414245000 8100077

      PAGE       PHYSICAL      MAPPING       INDEX CNT FLAGS
ffffea0000407d00 101f4000 ffff888007948751   414243  1 fffffc0080014
uptodate,lru,swapbacked
crash>
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
crash> vtop -c 2214 0x414244000
VIRTUAL     PHYSICAL
414244000   144ad000

   PGD: f2e0000 => 8608067
   PUD: 8608080 => 7851067
   PMD: 7851508 => ded0067
   PTE: ded0220 => 144ad867
  PAGE: 144ad000

  PTE      PHYSICAL  FLAGS
144ad867  144ad000  (PRESENT|RW|USER|ACCESSED|DIRTY)

      VMA           START       END     FLAGS FILE
ffff88800acd6640  414243000  414245000 8100077

      PAGE       PHYSICAL      MAPPING       INDEX CNT FLAGS
ffffea0000512b40 144ad000 ffff888007948751   414244  1 fffffc0080014
uptodate,lru,swapbacked
crash>
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
(gdb) monitor phys
(gdb) set *(unsigned long *)0xded0218=0x144ad867
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
crash> vtop -c 2214 0x414243000
VIRTUAL     PHYSICAL
414243000   144ad000

   PGD: f2e0000 => 8608067
   PUD: 8608080 => 7851067
   PMD: 7851508 => ded0067
   PTE: ded0218 => 144ad867
  PAGE: 144ad000

  PTE     PHYSICAL  FLAGS
144ad867  144ad000  (PRESENT|RW|USER|ACCESSED|DIRTY)

      VMA           START       END     FLAGS FILE
ffff88800acd6640  414243000  414245000 8100077

      PAGE        PHYSICAL      MAPPING       INDEX CNT FLAGS
ffffea0000512b40 144ad000 ffff888007948751   414244  1 fffffc0080014
uptodate,lru,swapbacked
crash>
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
crash> vtop -c 2214 0x414244000
VIRTUAL     PHYSICAL
414244000   144ad000

   PGD: f2e0000 => 8608067
   PUD: 8608080 => 7851067
   PMD: 7851508 => ded0067
   PTE: ded0220 => 144ad867
  PAGE: 144ad000

  PTE     PHYSICAL  FLAGS
144ad867  144ad000  (PRESENT|RW|USER|ACCESSED|DIRTY)

      VMA           START       END     FLAGS FILE
ffff88800acd6640  414243000  414245000 8100077

      PAGE        PHYSICAL      MAPPING       INDEX CNT FLAGS
ffffea0000512b40 144ad000 ffff888007948751   414244  1 fffffc0080014
uptodate,lru,swapbacked
crash>
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
$ ./code01
PID: 2214
address mapped: 0x414243000

writing to addresses

reading and writing to the addresses

A: BBBBBBBBBBBBBBBBBBBB
B: BBBBBBBBBBBBBBBBBBBB
addresses accessed
```

Resultado negativo

Caso 03

Entrada obsoleta na TLB (TLB *stale entry*)

Alterando a PTE via módulo *do kernel*.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
$ ./codes/code02
PID: 10612
address mapped: 0x414243000
A: AAAAAAAAAAAAAAAAAAAA
B: BBBBBBBBBBBBBBBBBBBB
Overwriting page table
check result
A: BBBBBBBBBBBBBBBBBBBB
B: BBBBBBBBBBBBBBBBBBBB
$
```

Resultado negativo, até executar o código diversas vezes

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
$ ./codes/code02
PID: 11057
address mapped: 0x414243000
A: AAAAAAAAAAAAAAAAAAAAA
B: BBBBBBBBBBBBBBBBBBBBB
Overwriting page table
pointer: 4242424242424242
$
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
$ ./codes/code02
PID: 11061
address mapped: 0x414243000
A: AAAAAAAAAAAAAAAAAAAA
B: BBBBBBBBBBBBBBBBBBBB
Overwriting page table
pointer: 4242424242424242
$
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
$ ./codes/code02
PID: 11065
address mapped: 0x414243000
A: AAAAAAAAAAAAAAAAAAAA
B: BBBBBBBBBBBBBBBBBBBB
Overwriting page table
pointer: 4141414141414141
$
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
$ ./codes/code02
PID: 11069
address mapped: 0x414243000
A: AAAAAAAAAAAAAAAAAAAA
B: BBBBBBBBBBBBBBBBBBBB
Overwriting page table
pointer: 4141414141414141
$
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

```
$ ./codes/code02
PID: 11073
address mapped: 0x414243000
A: AAAAAAAAAAAAAAAAAAAAA
B: BBBBBBBBBBBBBBBBBBBBB
Overwriting page table
pointer: 4242424242424242
$
```

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Resumo
  - Obtive resultados positivos em experimentos realizados acessando entradas obsoletas na TLB. Foram realizados diversos testes, incluindo acessando entradas obsoletas na TLB independente dos *bits* Present, RW, Dirty e outros na PTE.
  - Foi difícil, mas foi recompensador. Aprendi bastante, incluindo como eu dificultei muito a realização de alguns experimentos.
  - Nada substitui o trabalho manual e direto para obter conhecimento. Você pode ler 100 livros sobre determinado assunto, mas quando decidir experimentar com esse assunto diretamente, verá que a realidade é bem mais complexa.

Caso 03 - Entrada obsoleta na TLB (TLB *stale entry*)

- Lições
  - Tente não economizar tempo. Foque em algo solido. Em outras palavras, não seja preguiçoso (como eu fui).
  - Peça ajuda caso necessário. Não tenha medo e vergonha.
  - Consulte a documentação adequada e confirme que seu entendimento está de acordo. Cada palavra em um manual é importante.
    - Não foi o caso, mas existem diversos casos da documentação oficial estar errada, então é preciso cuidado, paciência e muita experiência para poder identificar esses casos.
    - Seja sincero com você, se não for experiente o suficiente, não ache que está descobrindo a vulnerabilidade mais interessante que já existiu. Pode acontecer, mas provavelmente é seu código, suposições, ambiente ou interpretação dos resultados que estão errados.
  - Experimente. Experimente. Experimente.

Caso 04

Comportamento não compreendido ao executar código via KVM

Caso 04 - Comportamento não compreendido ao executar código via KVM

- Alguns meses atrás, decidi aprender mais sobre virtualização e passei um tempo mexendo com KVM e VMware.
- Utilizei um artigo público como base:
  - https://zserge.com/posts/kvm/
- Após entender o básico e reproduzir o conteúdo do *blog post*, comecei a fazer alterações e estudar o comportamento.
  - Adicionando um `movw %REG,%cs`, o código se comporta de um jeito que eu não consegui entender e explicar.
- Assim, comecei um processo para entender melhor o que estava acontecendo.

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
_start:
   xor %ax,%ax
   xor %bx,%bx
loop:
   out %ax, $0x10
   inc %ax
   movw %bx,%ss
   jmp loop
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
$ sudo ./kvm_host_simple guest
IO port: 10, data: 0
IO port: 10, data: 1
IO port: 10, data: 2
IO port: 10, data: 3
IO port: 10, data: 4
IO port: 10, data: 5
IO port: 10, data: 6
IO port: 10, data: 7
IO port: 10, data: 8
IO port: 10, data: 9
IO port: 10, data: a
^C
$
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
_start:
   xor %ax,%ax
   xor %bx,%bx
loop:
   out %ax, $0x10
   inc %ax
   movw %bx,%cs
   jmp loop
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
$ sudo ./kvm_host_simple guest
IO port: 10, data: 0
IO port: 10, data: 0
IO port: 10, data: 0
IO port: 10, data: 0
IO port: 10, data: 0
IO port: 10, data: 0
IO port: 10, data: 0
IO port: 10, data: 0
IO port: 10, data: 0
IO port: 10, data: 0
IO port: 10, data: 0
^C
$
```

O que está acontecendo?

Caso 04 - Comportamento não compreendido ao executar código via KVM

- Hipótese inicial
  - O código parece estar sendo reiniciado. Por que?
- Realizei diversos testes para confirmar se o problema era realmente com CS ou com os valores sendo movidos.
  - Fiz testes com valores diferentes
  - Fiz testes com registradores diferentes
  - Fiz vários testes para confirmar que realmente o código estava “reiniciando”.
- BUG, *feature* ou vulnerabilidade?

Caso 04 - Comportamento não compreendido ao executar código via KVM

- Compreendo que `mov` para registrador CS não é permitido.
  - O registrador CS contém informações importantes, principalmente o CPL (*Current Privilege Level*) e não deve ser alterado de forma arbitrária. A modificação do registrador CS deve ser feita via os meios adequados, que inclui as instruções `syscall`/`sysret` ou `int`/`iret`.
  - Essa modificação gera uma exceção, #UD (*Undefined Instruction*), mas por qual motivo o código volta para o início?
- Comecei o processo de *debugging* e entender como uma #UD em um código no *guest* é gerenciado pelo KVM.

Caso 04 - Comportamento não compreendido ao executar código via KVM

“Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the “Operation” algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.”

https://www.felixcloutier.com/x86/mov

Caso 04 - Comportamento não compreendido ao executar código via KVM

- A função `handle_exception_nmi()` gerencia exceções. Tem uma validação realizada na função `is_invalid_opcode()`. Se ela retornar sucesso, a f`unção handle_ud()` é chamada.
- Em `handle_ud()`, checa por emulação.
  - Aqui descubro que KVM pode emular algumas instruções e a configuração `force_emulation_prefix`
- Executando o código e realizando *debugging* da execução, vejo que a função `x86_emulate_instruction()` é executada e começo a pensar que a instrução está sendo emulada.

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
_start:
    movw $0xc0d3,%ax
    out %ax,$0xff
    xorw %ax,%ax
    xorw %bx,%bx
    loop:
        out %ax, $0x10
        inc %ax
        movw %bx,%cs
        jmp loop
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
$ sudo ./kvm_host_simple guest
IO port: ff, data: c0d3
IO port: 10, data: 0
IO port: ff, data: c0d3
IO port: 10, data: 0
IO port: ff, data: c0d3
IO port: 10, data: 0
IO port: ff, data: c0d3
IO port: 10, data: 0
IO port: ff, data: c0d3
IO port: 10, data: 0
IO port: ff, data: c0d3
^C
$
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

- Confirmo que o código de emulação do KVM está recebendo a instrução do código da VM *guest* corretamente e várias outras informações para garantir que a execução do emulador no KVM realmente está no contexto do código do *guest*.
  - É realmente o opcode do `movw %REG,%cs` e o IP (*Instruction Pointer*) está de acordo.
- Após alguns minutos, penso melhor e vejo que a emulação não é realmente executada. Ela retorna erro.
  - A função `kvm_queue_exception()` é executada.
- Neste ponto, já tenho evidências que uma exceção #UD vai ser injetada na máquina virtual *guest*, mas até o momento isso não explica o comportamento.
  - Aprendo que a exceção vai ser injetada e antes do código re-executar a VM, o KVM checa se tem uma exceção.

Caso 04 - Comportamento não compreendido ao executar código via KVM

- Consigo ver que a exceção está sendo entregue à VM através do VMCS (*Virtual Machine Control Structure*) através da função vmx_inject_exception().

```
1760            if (kvm_exception_is_soft(ex->vector)) {
1761                    vmcs_write32(VM_ENTRY_INSTRUCTION_LEN,
vmx->vcpu.arch.event_exit_inst_len);
1763                    intr_info |= INTR_TYPE_SOFT_EXCEPTION;
1764            } else
1765                    intr_info |= INTR_TYPE_HARD_EXCEPTION;
...
1767            vmcs_write32(VM_ENTRY_INTR_INFO_FIELD,
intr_info);
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
(gdb) print/x vcpu->arch.exception
$9 = {
  pending = 0x1,
  injected = 0x0,
  has_error_code = 0x0,
  vector = 0x6,
  error_code = 0x0,
  payload = 0x0,
  has_payload = 0x0
}
(gdb)
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
17 #define DE_VECTOR 0
18 #define DB_VECTOR 1
19 #define BP_VECTOR 3
20 #define OF_VECTOR 4
21 #define BR_VECTOR 5
22 #define UD_VECTOR 6
23 #define NM_VECTOR 7
24 #define DF_VECTOR 8
25 #define TS_VECTOR 10
26 #define NP_VECTOR 11
27 #define SS_VECTOR 12
28 #define GP_VECTOR 13
29 #define PF_VECTOR 14
30 #define MF_VECTOR 16
31 #define AC_VECTOR 17
32 #define MC_VECTOR 18
33 #define XM_VECTOR 19
34 #define VE_VECTOR 20
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

- #UD
  - Após a confirmação de que o componente responsável pelo comportamento é a própria VM (a exceção está sendo entregue a ela), comecei a entender que o comportamento é normal, esperado, e comecei a pesquisar como uma #UD é gerenciada em *real mode*.
  - Ainda não entendia como *real mode* era implementado no KVM. Encontrei uma *thread* em uma lista de discussão (qemu-devel) que menciona algumas diferenças fundamentais:
    - Em determinado momento do passado, uma CPU não poderia operar realmente em *real mode* e o modo vm8086 era utilizado para fornecer essa capacidade, porém, as CPUs modernas podem operar em "*real mode* virtualizado" (*unrestricted guest mode*).

Caso 04 - Comportamento não compreendido ao executar código via KVM

- #UD
  - Ainda não entendia como *real mode* era implementado no KVM. Encontrei uma *thread* em uma lista de discussão (qemu-devel) que menciona algumas diferenças fundamentais:
    - Na *thread* também ficou claro que as exceções são injetadas/entregues/agendadas para a máquina virtual rodando em *real mode*. Nesse momento percebi que o comportamento se deve a algo na memória ou no código da VM *guest*.
    - Questions about the real mode in kvm/qemu
      https://lore.kernel.org/qemu-devel/CAKXe6SL42HeidUzQG_8JkutgRtO8AtC08OHsQRPnVH9POTubAw@mail.gmail.com/t/#m7bb84d82558dadc4a219414c8417c8c30137b8fb

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
On 26/09/19 09:52, Li Qiang wrote:
> Hi Paolo and all,
>
> There are some question about the emulation for real mode in kvm/qemu.
> For all the
> question I suppose the 'unstrict guest' is not enabled.
>
> 1. how the protected mode CPU emulate the real mode? It seems it uses
> vm86, however, vm86 is not available in x86_64 CPU? So what's the
> 'to_vmx(vcpu)->rmode.vm86_active' here vm86 means?

vm86 mode is available in 64-bit CPUs; it is not available in 64-bit
mode (EFER.LMA=1).

Once KVM places the processor in VMX non-root mode (VMLAUNCH/VMRESUME),
the processor can leave 64-bit mode and run in vm86 mode.  And that's
exactly what KVM does on processors without unrestricted guest support.

In that mode, KVM will start trapping all exceptions and send them to
handle_rmode_exception.  This in turn will either emulate privileged
instructions (for #GP or #SS) or inject them as realmode exceptions.
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

- Após ler a conversa na lista de discussão, ficou claro que o comportamento é explicado por componentes da VM *guest*. A memória, o código ou a arquitetura em execução.
  - Sendo assim, revisitei como as exceções são gerenciadas em *real mode*.
- Segui com a hipótese de que a tabela de exceções (IDT ou IVT) é inicializada no endereço físico 0 e que a entrada da #UD estava sendo populada com zeros, fazendo com que o *interrupt handler* apontasse para o endereço 0.
  - Esse comportamento explicaria a ilusão de que o código estava sendo reinicializado.
  - O comportamento acontece porque o código executado na VM *guest* era pequeno e não chegava a popular a entrada do #UD na tabela de exceções.
- Realizei um teste para confirmar a hipótese populando a memória com mais código.
  - Como resultado, o código não mais retorna para o início. Ele fica parecendo que está travado (*stuck*).

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
_start:
    mov    $0xc0d3,%ax
    out    %ax,$0xff
    xor    %ax,%ax
    loop:
        out    %ax,$0x10
        inc    %ax
        mov    $0xffff,%cx
        mov    %cx,%cs
        jmp    loop
        .string “AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA”
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
$ sudo ./kvm_host_simple guest
IO port: ff, data: c0d3
IO port: 10, data: 0
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

- Próximo passo é controlar corretamente qual código vai executar quando uma #UD acontecer. Alterei o *hypervisor* para inserir o código do *guest* no *offset* 1024 da memória (deixando os primeiros 1024 *bytes* sem uso).
  - Dessa forma, podemos popular corretamente as entradas da tabela de exceções e definir quais funções serão executadas em determinadas situações.
    - Criei um *handler* no código que executa a instrução `hlt` quando um #UD é gerado.
    - Configurei a VM para gerar um VM-exit quando a instrução `hlt` é executada. O *hypervisor* detecta a instrução hlt e finaliza a VM.

Caso 04 - Comportamento não compreendido ao executar código via KVM

```
$ sudo ./kvm_host_simple guest
IO port: ff, data: c0d3
IO port: 10, data: 0
hlt
Exiting VM
$
```

Caso 04 - Comportamento não compreendido ao executar código via KVM

"9.7.1 Real-Address Mode IDT In real-address mode, the only system data structure that must be loaded into memory is the IDT (also called the "interrupt vector table"). By default, the address of the base of the IDT is **physical address 0H**. This address can be changed by using the LIDT instruction to change the base address value in the IDTR. Software initialization code needs to load interrupt- and exception-handler pointers into the IDT before interrupts can be enabled."

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide - 11.7.1 Real-Address Mode IDT

Caso 04 - Comportamento não compreendido ao executar código via KVM

- Resumo
  - Levei um dia (até para minha surpresa) para entender o comportamento.
  - Criei hipóteses que não estavam de acordo, porém procurei validar e entender o comportamento antes de assumir com certeza.
  - O código é complexo. Virtualização, *kernel*, KVM, VMware, *debugging*, mas mesmo assim foi possível entender com precisão o que estava acontecendo e até alterar o comportamento.
  - O aprendizado foi de grande valia.

Caso 04 - Comportamento não compreendido ao executar código via KVM

- Lições
  - Entender a causa raiz
  - Criar hipóteses é parte fundamental do trabalho, porém precisamos ter cuidado.
  - Calma e paciência.
  - Procurar referências externas dos pontos levantados.
  - Validação das suposições é fundamental. É muito comum assumirmos um determinado comportamento.
  - A realidade é complexa. Pode enganar facilmente. Nem sempre o que estamos vendo (interpretando) é realmente o que está acontecendo.
  - Acontecem muitas coincidências. O profissional precisa ser capaz de identificá-las corretamente e não se iludir.

Conclusão

Conclusão

- Um processo metodológico rigoroso fortalece a compreensão e entendimento dos objetos de estudos. Reduz as armadilhas, tanto do nosso comportamento quanto dos sistemas. Torna sólido e robusto o estudo e a compreensão de fenômenos.
- Otimiza o resultado. Facilita a compreensão, o entendimento, escrita de documentação, modificações e inclusive, a explicação à outros.
- Com uma mente clara, deve se tornar mais fácil o mapeamento dos possíveis problemas e soluções.
- Estude ciência!

MUITO OBRIGADO!

Dúvidas?