NSFOCUS

TECHWORLD2019

绿 盟 科 技 技 术 嘉 年 华

探索 · DISCOVERY

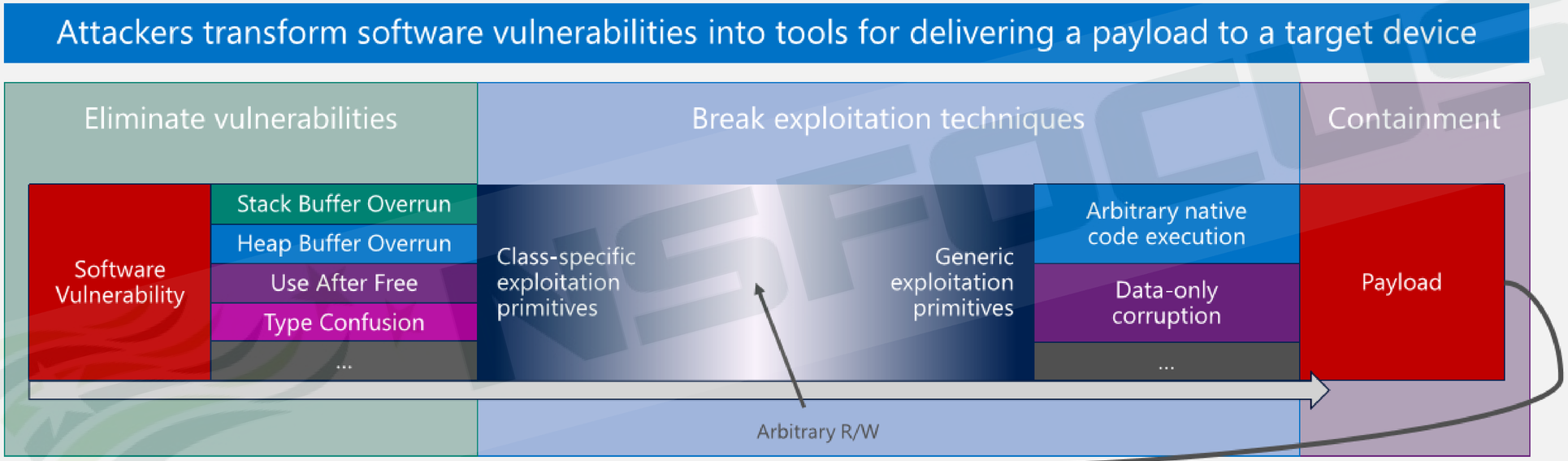
NSFOCUS TECHWORLD

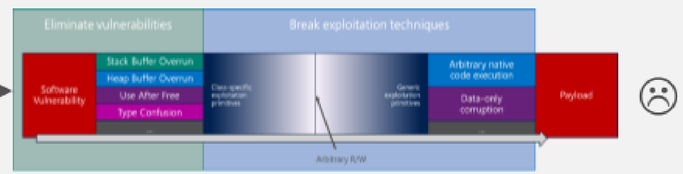
Liar Game

The Secret of Mitigation Bypass Techniques

缓解措施是什么

How we think about mitigating software vulnerabilities





Attackers typically need to elevate privileges

This means applying the same defenses to privileged attack surfaces

At some point, we lose containment as a defense

This leaves eliminating vulnerabilities & breaking techniques

缓解措施是什么

Technologies for mitigating code execution

Prevent arbitrary code generation

Code Integrity Guard

Images must be signed and load from valid places

- ✓ Only valid, signed code pages can be mapped by the app

Arbitrary Code Guard

Prevent dynamic code generation, modification, and execution

- ✓ Code pages are immutable and cannot be modified by the app

Prevent control-flow hijacking

Control Flow Guard

Enforce control flow integrity on indirect function calls

- ✓ Code execution stays "on the rails" per the control-flow integrity policy

???

Enforce control flow integrity on function returns

缓解措施如何工作

机制

```
if (is_allowed_by_mitigation_policy()) {
        do_sensitive_action();
} else {
        fail_fast();
}
```

假设

缓解措施能有效工作的前提条件

操作系统能正常工作的例外规则

缓解措施：DEP

机制

```
if (PTE(address).NX == 0) {
      execute(address);
} else {
      fail_fast();
}
```

假设

代码段中的代码是可信的

严格遵守 W^X 原则

代码段中的代码都是可信的吗?

```
f3 0f 59 c3 f3 0f 58 86-8c 00 00 00 f3 0f 11 86

f30f59c3       mulss   xmm0,xmm3
f30f58868c000000 addss   xmm0,dword ptr [rsi+8Ch]

59             pop     rcx
c3             ret
```

严格遵守 W^X 原则

- 避免使用可读写执行 (PAGE_EXECUTE_READWRITE) 内存
- 在内存的整个生命周期中保持 W^X

ATL Thunk Pool 问题

- 函数 __AllocStdCallThunk_cmn 会分配可读写执行内存用于保存 Thunk

```
mem = VirtualAlloc(0i64, 0x1000ui64, 0x1000u, 0x40u);
if ( !mem )
  return 0i64;
next = *mem;
thunk = InterlockedPopEntrySList(__AtlThunkPool);
if ( thunk )
{
  VirtualFree(mem, 0i64, 0x8000u);
}
else
{
  end = mem + 0xFE0;
  do
  {
    InterlockedPushEntrySList(__AtlThunkPool, mem);
    mem += 0x20;
  }
  while ( mem < end );
  thunk = mem;
}
return thunk;
```

ATL Thunk Pool 问题修复

- 引入 atlthunk.dll 实现数据与代码的分离
  - AtlThunk_AllocateData
  - AtlThunk_InitData
  - AtlThunk_DataToCode
  - AtlThunk_FreeData

```
FARPROC __fastcall GetProcAddressAll_AtlThunkData()
{
  HMODULE atlthunk; // rax MAPDST
  int v1; // [rsp+0h] [rbp-28h]

  if ( loaded )
    return DecodePointer(AllocateData);
  atlthunk = LoadLibraryExA("atlthunk.dll", 0i64, 0x800u);
  if ( atlthunk
    && GetProcAddressSingle(atlthunk, "AtlThunk_AllocateData", &AllocateData)
    && GetProcAddressSingle(atlthunk, "AtlThunk_InitData", &InitData)
    && GetProcAddressSingle(atlthunk, "AtlThunk_DataToCode", &DataToCode)
    && GetProcAddressSingle(atlthunk, "AtlThunk_FreeData", &FreeData) )
  {
    _InterlockedOr(&v1, 0);
    loaded = 1;
    return DecodePointer(AllocateData);
  }
  return 0i64;
}
```

ATL Thunk Pool 问题修复

- 用函数 AtlThunk_AllocateData 代替函数 __AllocStdCallThunk_cmn 来分配 Thunk

```
ThunkData *AtlThunk_AllocateData()
{
  HANDLE heap; // rax MAPDST
  ThunkData *data; // rbx
  __int64 (*AllocateData)(void); // rax
  Thunk *Thunk; // rax

  heap = GetProcessHeap();
  data = HeapAlloc(heap, 8u, 0x10ui64);
  if ( data )
  {
    AllocateData = GetProcAddressAll_AtlThunkData();
    data->fallbck = AllocateData == 0i64;
    if ( AllocateData )
      Thunk = AllocateData();
    else
      Thunk = __AllocStdCallThunk_cmn();
    data->thunk = Thunk;
    if ( Thunk )
      return data;
    heap = GetProcessHeap();
    HeapFree(heap, 0, data);
  }
  return 0i64;
}
```

ATL Thunk Pool 问题修复

兼容性处理

- 新控件在新系统中
  - 调用函数 AtlThunk_AllocateData
- 新控件在旧系统中
  - 调用函数 __AllocStdCallThunk_cmn
- 旧控件在新系统中
  - 调用函数 __AllocStdCallThunk_cmn

ATL Thunk Pool 回退攻击

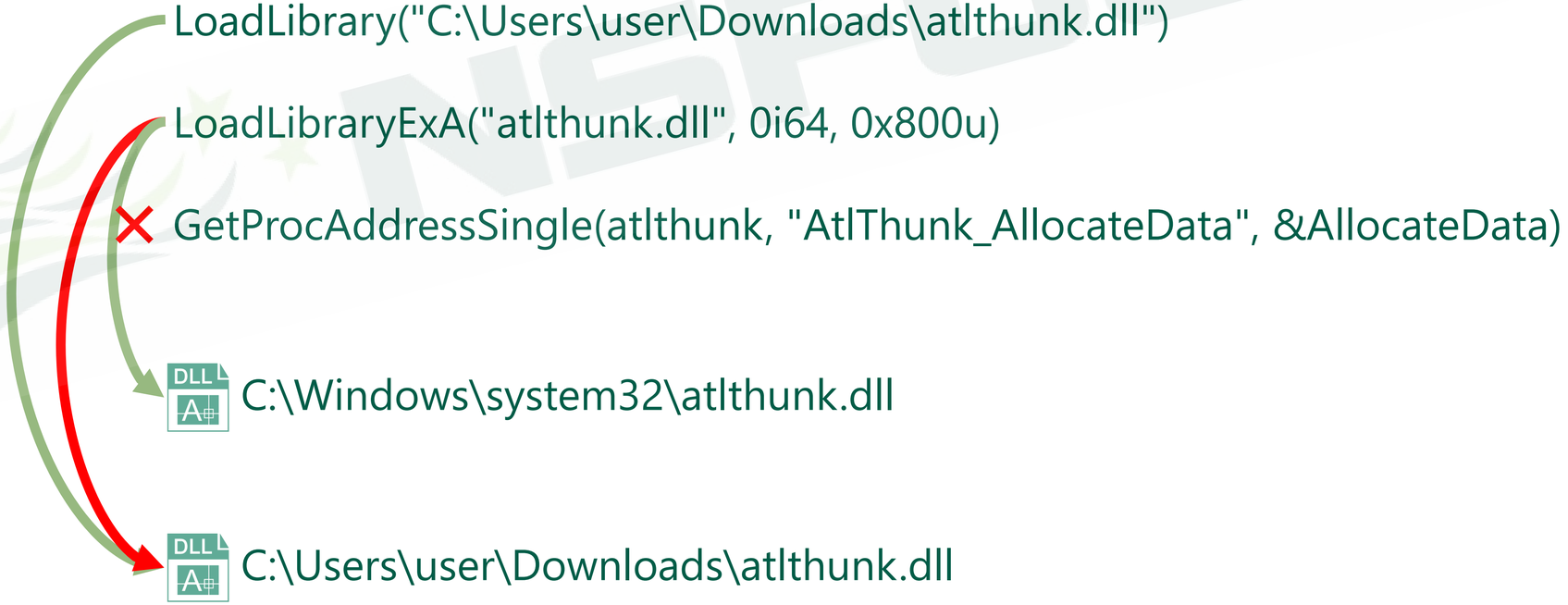
修复方案的假设

- 调用函数 GetProcAddressAll_AtlThunkData 成功
  - LoadLibraryExA("atlthunk.dll", 0i64, 0x800u) 成功
  - GetProcAddressSingle(atlthunk, "AtlThunk_AllocateData", &AllocateData) 成功
  - GetProcAddressSingle(atlthunk, "AtlThunk_InitData", &InitData) 成功
  - GetProcAddressSingle(atlthunk, "AtlThunk_DataToCode", &DataToCode) 成功
  - GetProcAddressSingle(atlthunk, "AtlThunk_FreeData", &FreeData) 成功

ATL Thunk Pool 回退攻击

修复方案的假设

- 调用函数 GetProcAddressAll_AtlThunkData 成功

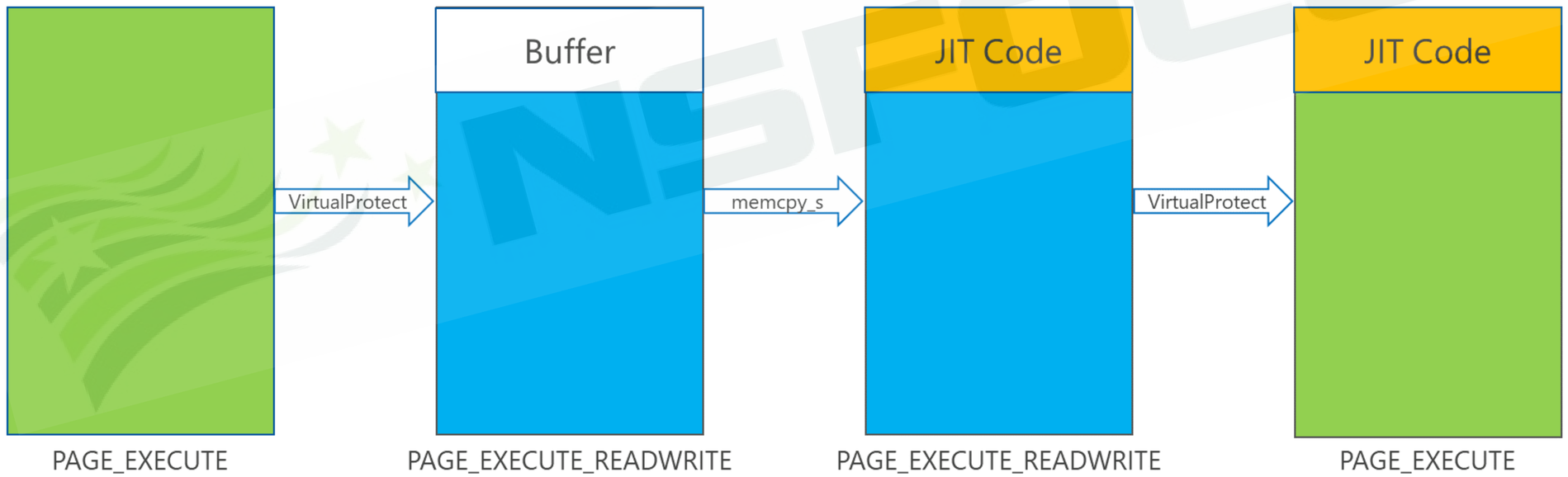


JIT 编译问题

- 主流浏览器已经做到在 JIT 编译时避免使用可读写执行内存
  - Microsoft Edge 不常驻可读写执行内存
  - Firefox 从 46.0 开始不常驻可读写执行内存
  - Chrome 从 64.0 开始不常驻可读写执行内存

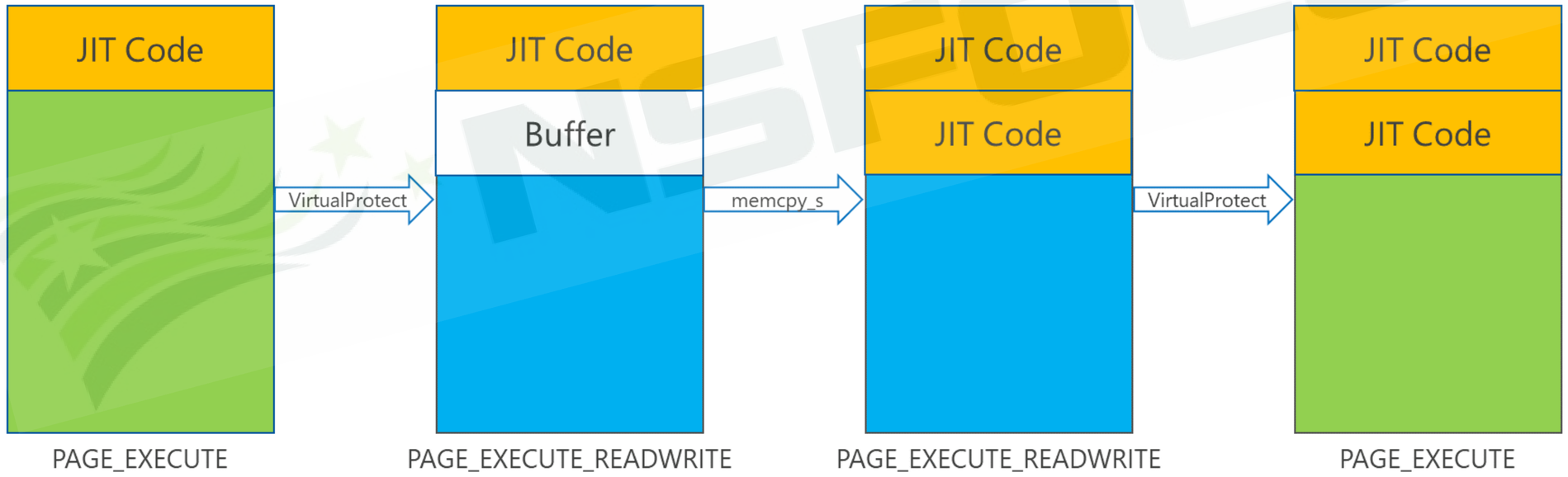
JIT 编译问题

JIT 编译如何使用内存



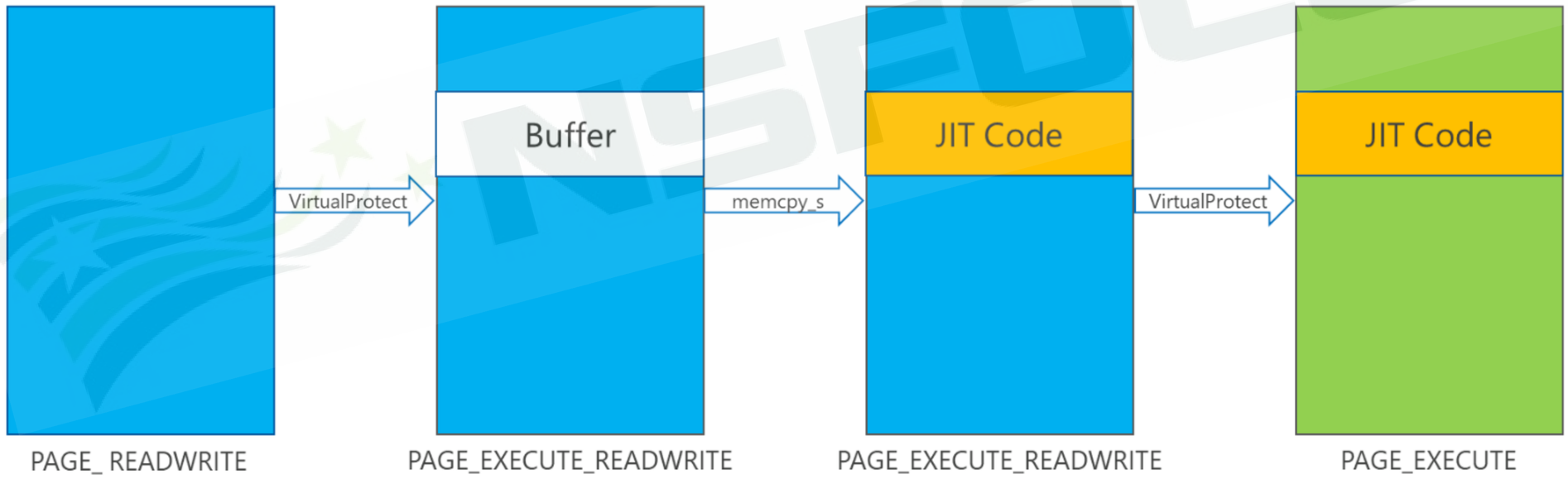
JIT 编译问题

JIT 编译如何使用内存



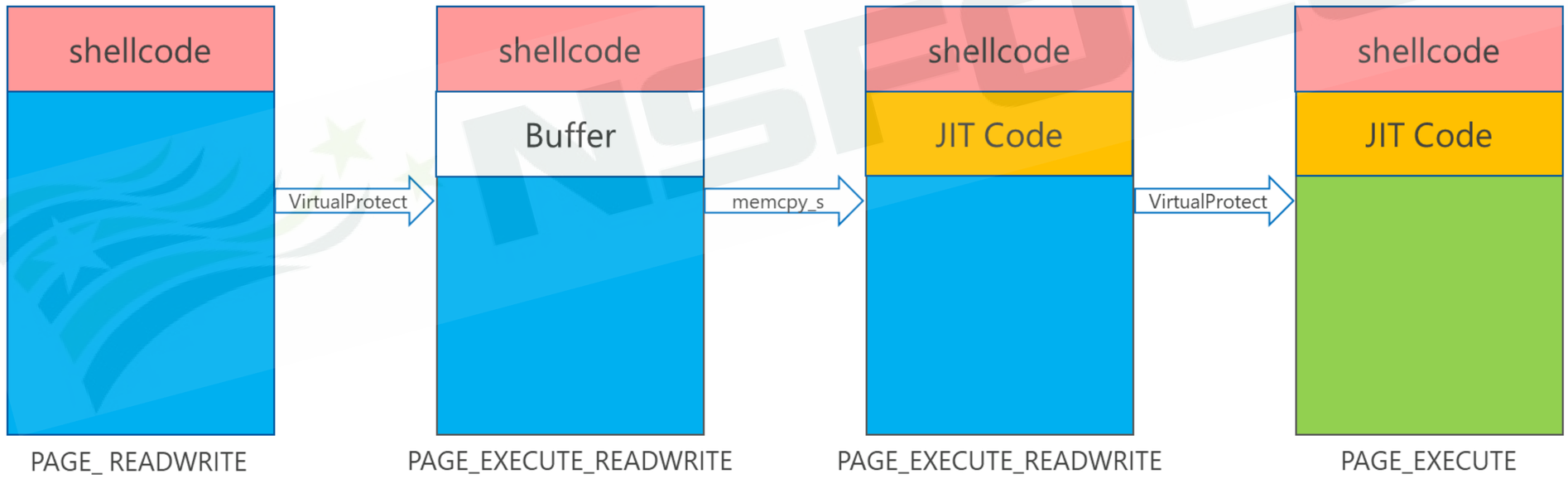
JIT 编译问题

- 欺骗浏览器替换 JIT 编译使用的内存



JIT 编译问题

- 事先写入的数据将变为可执行的代码



缓解措施：CFG

机制

```
if (CFG_Bitmap[address] == 1) {
      call(address);
} else {
      fail_fast();
}
```

假设

CFG Bitmap 中置位的地址是可信的

CFG 使用的指针是可信的

CFG Bitmap 中置位的地址都是可信的吗?

- 未启用 CFG 的模块
- 导出函数
- JIT 编译生成的代码

CFG Bitmap 中置位的地址都是可信的吗?

- 未启用 CFG 的模块
  - CFG Bitmap 中所有对应位都被置位

CFG Bitmap 中置位的地址都是可信的吗?

- 未启用 CFG 的模块
  - CFG Strict Mode 阻止加载未启用 CFG 的模块

CFG Bitmap 中置位的地址都是可信的吗?

- 导出函数
  - CFG Bitmap 中导出函数对应位会置位
  - 敏感的导出函数
    - NtContinue
    - WinExec
    - LoadLibrary
    - ...

CFG Bitmap 中置位的地址都是可信的吗?

- 导出函数
  - CFG Export Suppression 在一定程度上解决导出函数问题

CFG Bitmap 中置位的地址都是可信的吗?

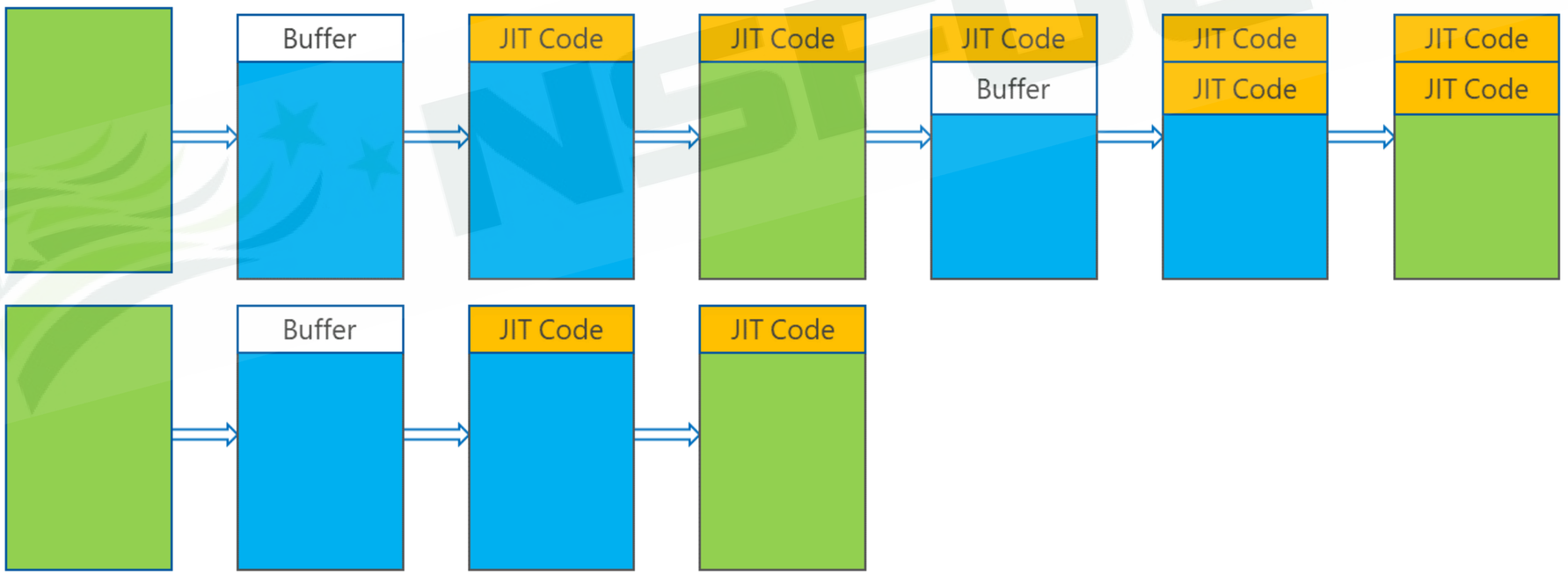
- JIT 编译生成的代码
  - 分配可执行内存或变更为可执行内存时默认会将 CFG Bitmap 中所有对应位置位

CFG Bitmap 中置位的地址都是可信的吗?

- JIT 编译生成的代码
  - 通过设置 PAGE_TARGETS_NO_UPDATE 来禁止置位
  - 显示调用 SetProcessValidCallTargets 进行置位

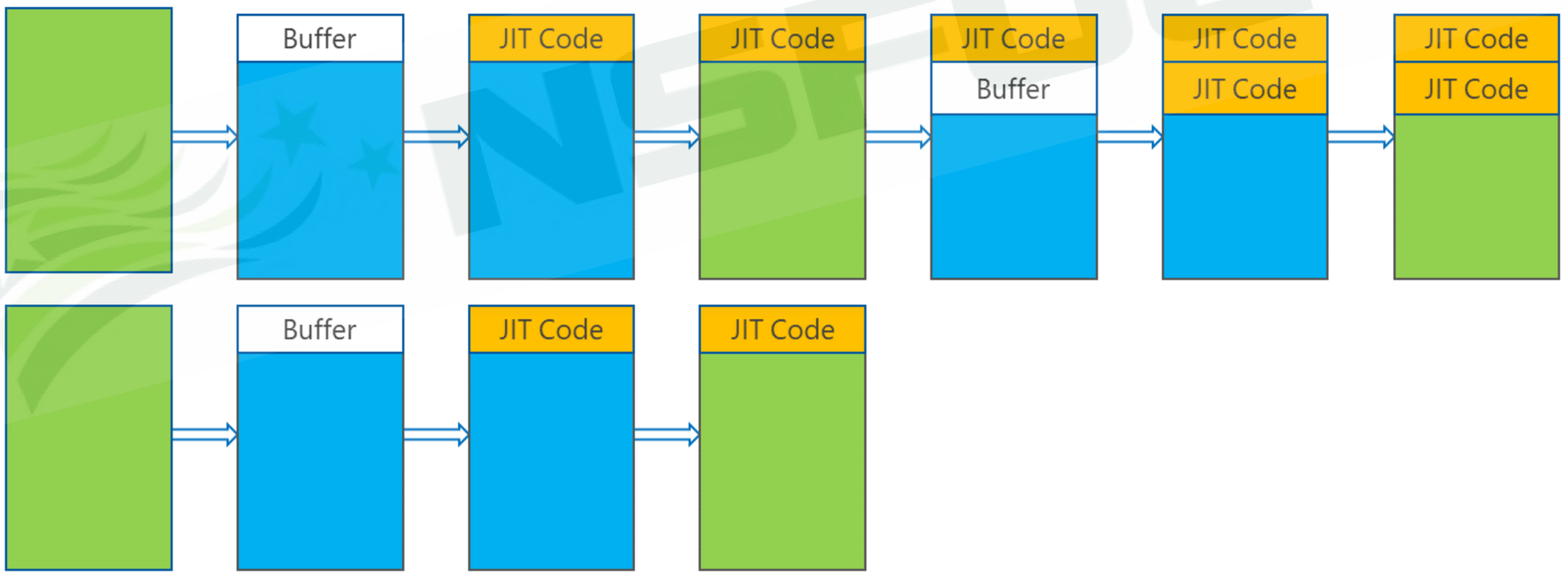
CFG Bitmap 中置位的地址都是可信的吗?

- 创建两个 JavaScript 引擎进行 JIT 编译



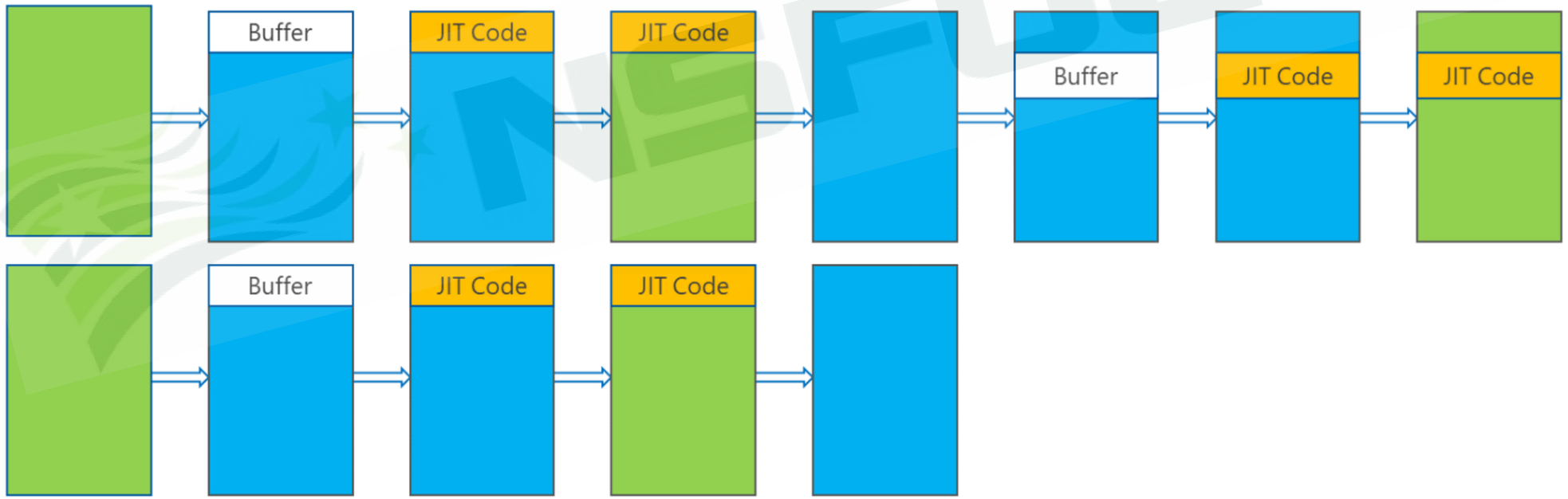
CFG Bitmap 中置位的地址都是可信的吗?

- 释放其中一个引擎，其使用的内存将变更为可读写



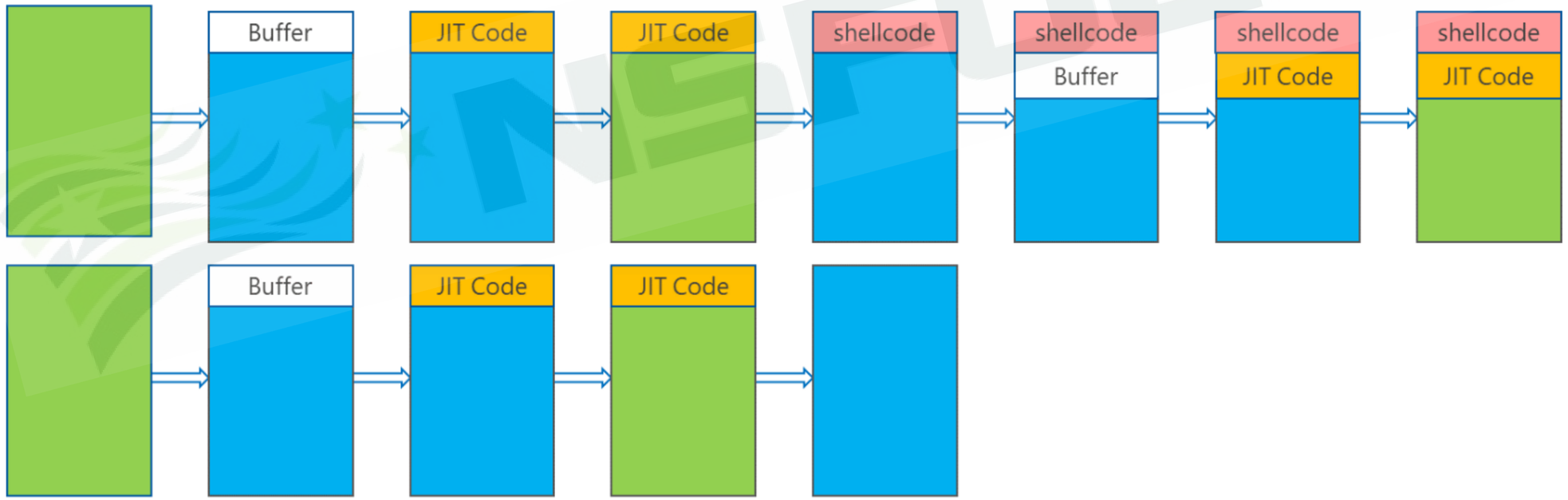
CFG Bitmap 中置位的地址都是可信的吗?

- 欺骗浏览器让两个引擎使用同一内存



CFG Bitmap 中置位的地址都是可信的吗?

- 事先写入的数据将变为可执行的代码，并且 CFG Bitmap 中有置位



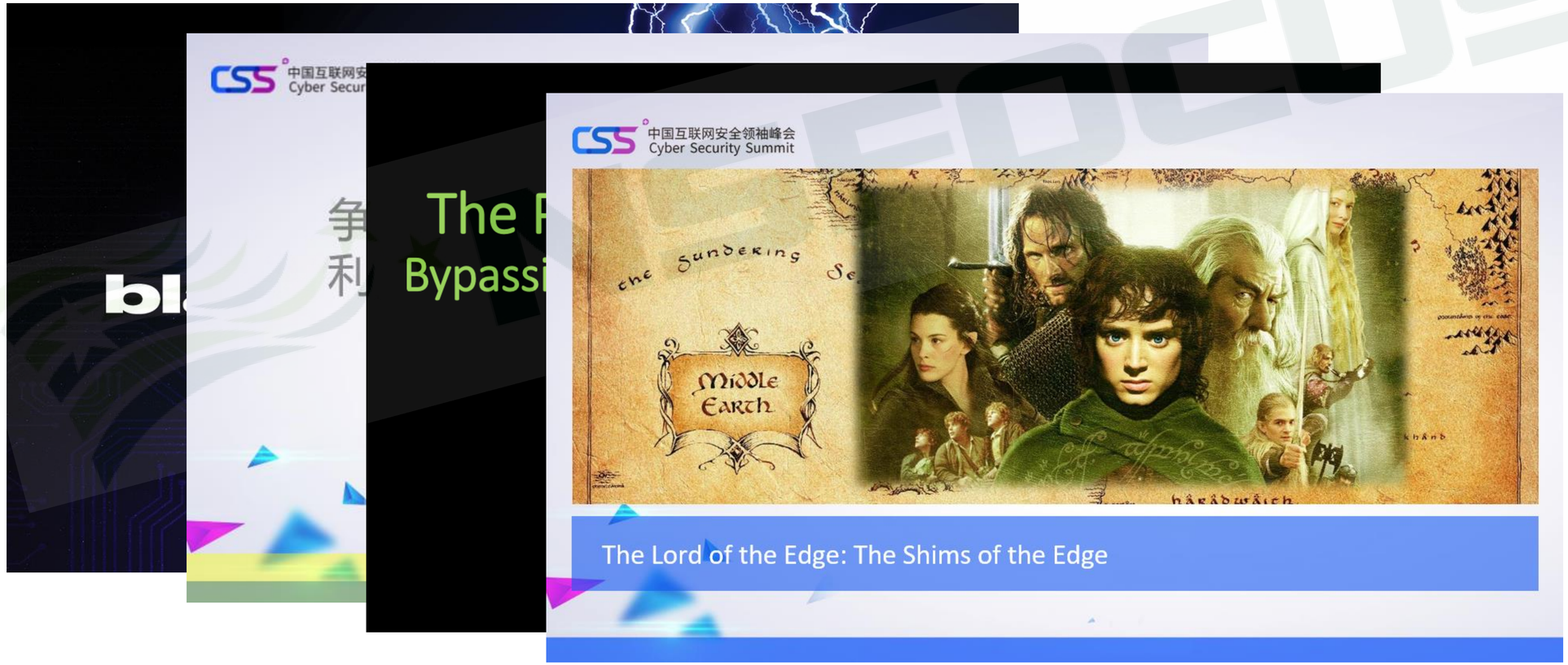
CFG 使用的指针都是可信的吗?

- 关键指针仅仅通过只读进行保护
  - guard_check_icall_fptr
  - __guard_dispatch_icall_fptr

```
.rdata:00000001805B42F8 ; ===========================================================================
.rdata:00000001805B42F8
.rdata:00000001805B42F8 ; Segment type: Pure data
.rdata:00000001805B42F8 ; Segment permissions: Read
.rdata:00000001805B42F8 _rdata          segment para public 'DATA' use64
.rdata:00000001805B42F8                 assume cs:_rdata
.rdata:00000001805B42F8                 ;org 1805B42F8h
.rdata:00000001805B42F8 __guard_check_icall_fptr dq offset Js::JavascriptFunction::CheckAlignment(void)
.rdata:00000001805B42F8                                         ; DATA XREF: IR::GetNonTableMethodAddre
.rdata:00000001805B42F8                                         ; Js::InterpreterStackFrame::DelayDynam
.rdata:00000001805B4300 __guard_dispatch_icall_fptr dq offset _guard_dispatch_icall_nop
```

CFG 使用的指针都是可信的吗?

- 欺骗系统来修改只读内存并不困难



缓解措施：ACG

机制

```
if (W^X(address, flNewProtect)) {
      change_protection(address, flNewProtect);
} else {
      fail_fast();
}
```

假设

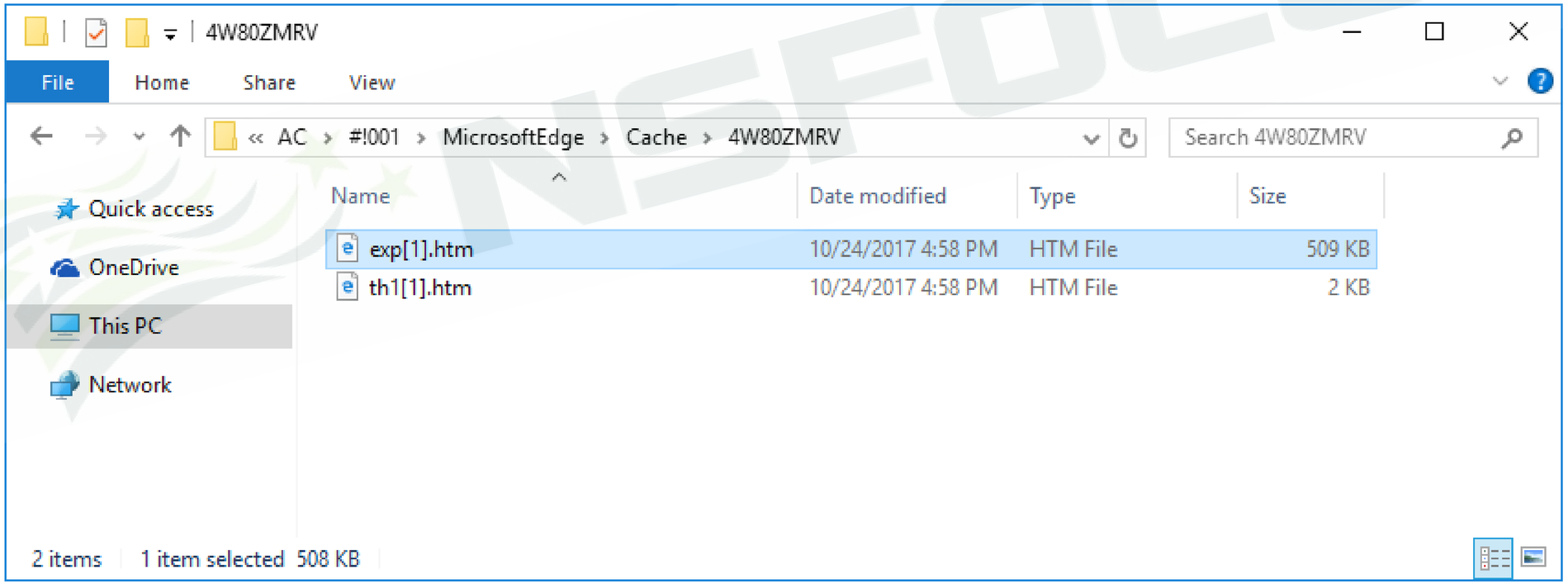
加载动态链接库时例外

可以加载任意动态链接库么？

- NoRemoteImages 阻止加载远程文件

可以加载任意动态链接库么？

- 利用浏览器缓存将动态链接库保存到本地后加载



缓解措施：CIG

机制

```
if (is_signed_by_microsoft(file)) {
      create_section(file);
} else {
      fail_fast();
}
```

假设

微软签名的动态链接库是可信的

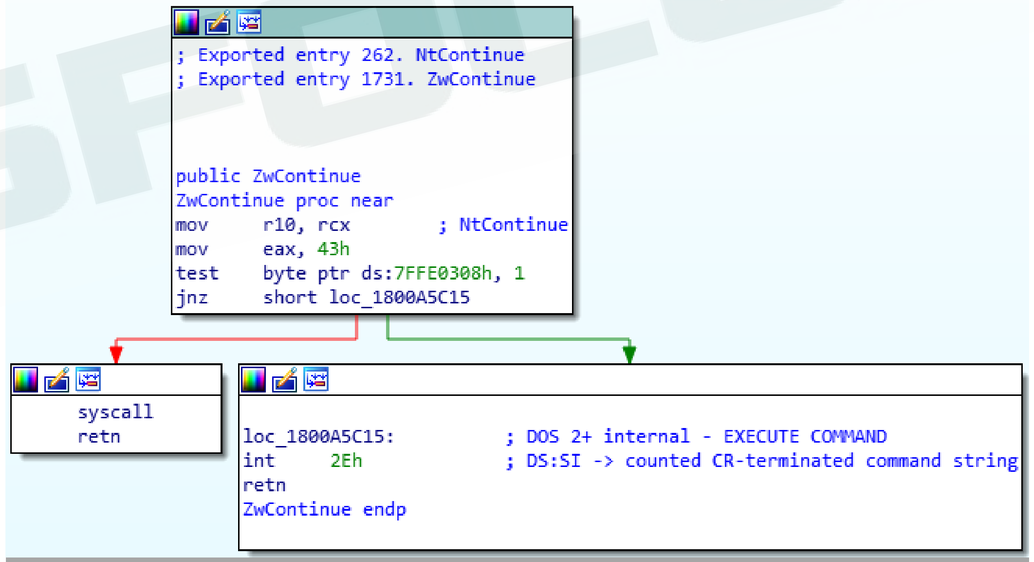
微软签名的动态链接库是可信的吗?

系统调用的版本差异

```
; Exported entry 430. NtQueryDefaultUILanguage
; Exported entry 1811. ZwQueryDefaultUILanguage


public ZwQueryDefaultUILanguage
ZwQueryDefaultUILanguage proc near
mov     r10, rcx        ; NtQueryDefaultUILanguage
mov     eax, 43h
syscall                 ; Low latency system call
retn
ZwQueryDefaultUILanguage endp
```

ntdll.dll version 6.3.9600.17936



ntdll.dll version 10.0.15063.0

微软签名的动态链接库是可信的吗?

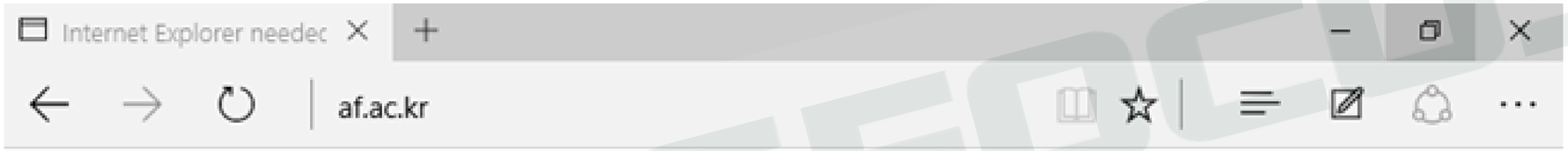
- 用旧版的 ntdll.dll 来欺骗系统
  - 调用 6.3.9600.17936 的 NtQueryDefaultUILanguage
  - 等同于调用 10.0.15063.0 的 NtContinue

高维欺骗技术

- 不直接与缓解措施进行对抗
- 通过伪造环境来滥用系统功能

高维欺骗技术

Launch IE



This website needs Internet Explorer

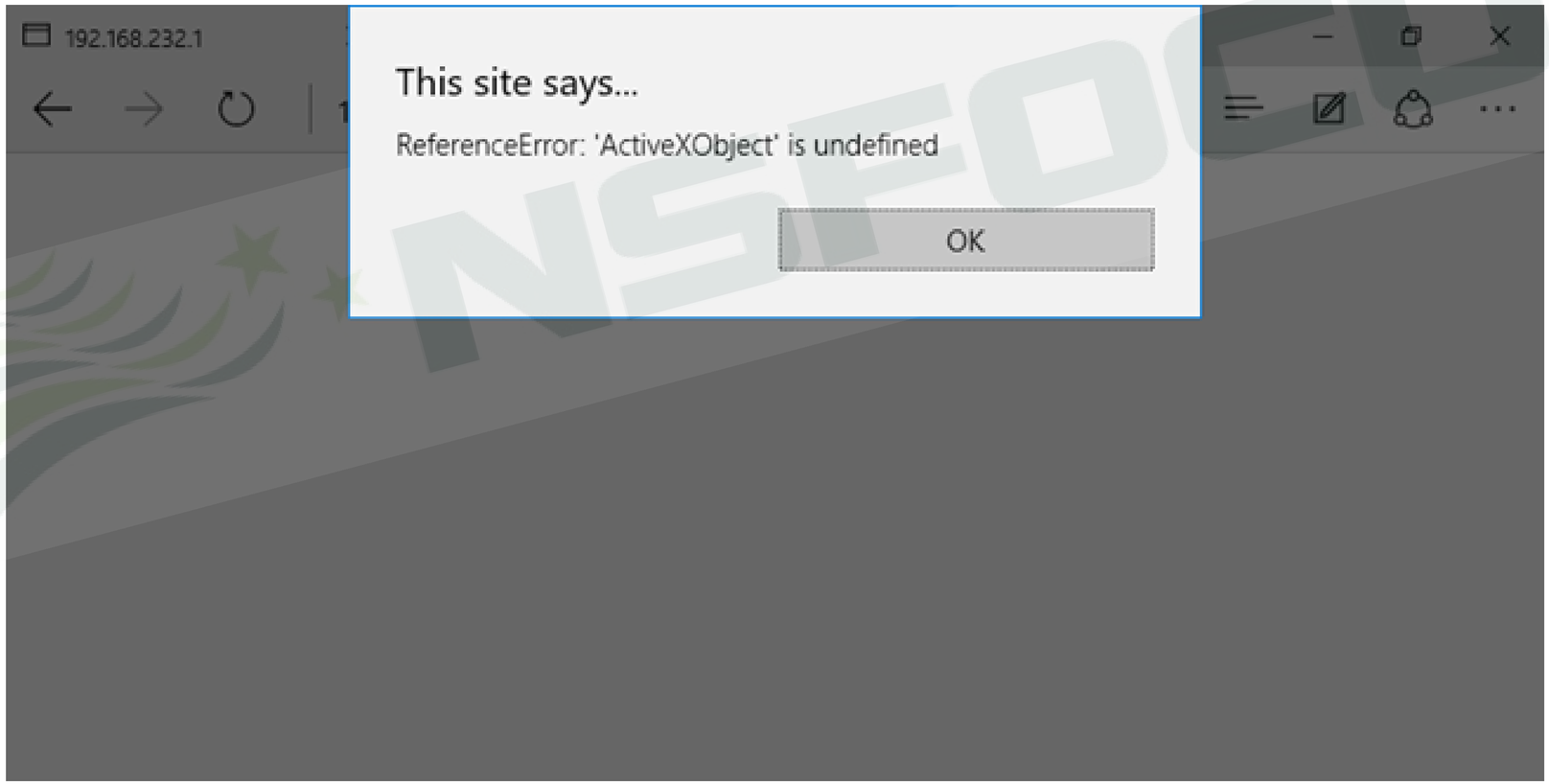
This website uses technology that will work best in Internet Explorer.

Open with Internet Explorer

Keep going in Microsoft Edge

高维欺骗技术

ActiveX



Technologies for mitigating code execution

Prevent arbitrary code generation

Code Integrity Guard

Images must be signed and load from valid places

Arbitrary Code Guard

Prevent dynamic code generation, modification, and execution

Prevent control-flow hijacking

Control Flow Guard

Enforce control flow integrity on indirect function calls

???

Enforce control flow integrity on function returns

- ✓ Only valid, signed code pages can be mapped by the app
- ✓ Code pages are immutable and cannot be modified by the app
- ✓ Code execution stays "on the rails" per the control-flow integrity policy

Technologies for mitigating code execution

Prevent arbitrary code generation

Code Integrity Guard

Images must be signed and load from valid places

Arbitrary Code Guard

Prevent dynamic code generation, modification, and execution

Prevent control-flow hijacking

Control Flow Guard

Enforce control flow integrity on indirect function calls

Fine Grained CFI

- ✓ Only valid, signed code pages can be mapped by the app
- ✓ Code pages are immutable and cannot be modified by the app
- ✓ Code execution stays "on the rails" per the control-flow integrity policy

Fine Grained CFI 并不是银弹

- Fine Grained CFI 的实现中也必然存在假设
- 如何保证这些假设的不变性是关键点

只读内存问题

- 缺少真正的只读内存
- 对关键数据的保护并不可靠

高维欺骗技术

NSFOCUS

TECHWORLD2019

绿 盟 科 技 技 术 嘉 年 华

探索 · DISCOVERY

NSFOCUS TECHWORLD

感谢聆听！