Xcon | XAUTOCAR

XCON XFOCUS INFORMATION SECURITY CONFERENCE

2017
8.23-24

XCON安全焦点
信息安全技术峰会

Fixed, or not fixed, that is the question

张云海

XCON XFOCUS INFORMATION SECURITY CONFERENCE

目录

CONTENTS

01 背景

02 利用 ATL Thunk Pool

03 利用 Chakra JIT Engine

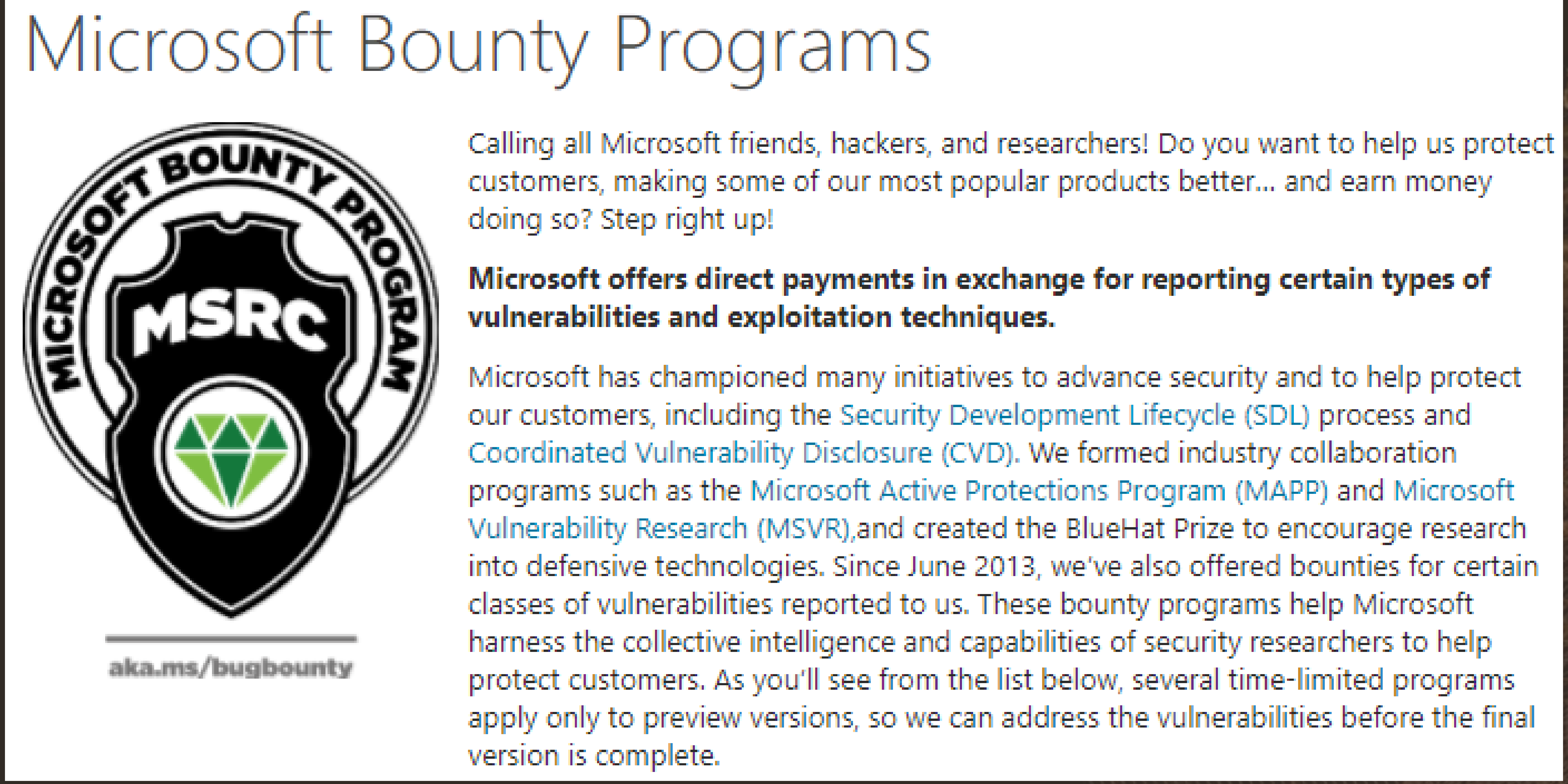
01

背景

- 缓解措施 – 5 年前

- 缓解措施 – 当前

- **微软在 2013 年启动了赏金计划**



Microsoft Bounty Programs

Calling all Microsoft friends, hackers, and researchers! Do you want to help us protect customers, making some of our most popular products better... and earn money doing so? Step right up!

Microsoft offers direct payments in exchange for reporting certain types of vulnerabilities and exploitation techniques.

Microsoft has championed many initiatives to advance security and to help protect our customers, including the Security Development Lifecycle (SDL) process and Coordinated Vulnerability Disclosure (CVD). We formed industry collaboration programs such as the Microsoft Active Protections Program (MAPP) and Microsoft Vulnerability Research (MSVR),and created the BlueHat Prize to encourage research into defensive technologies. Since June 2013, we've also offered bounties for certain classes of vulnerabilities reported to us. These bounty programs help Microsoft harness the collective intelligence and capabilities of security researchers to help protect customers. As you'll see from the list below, several time-limited programs apply only to preview versions, so we can address the vulnerabilities before the final version is complete.

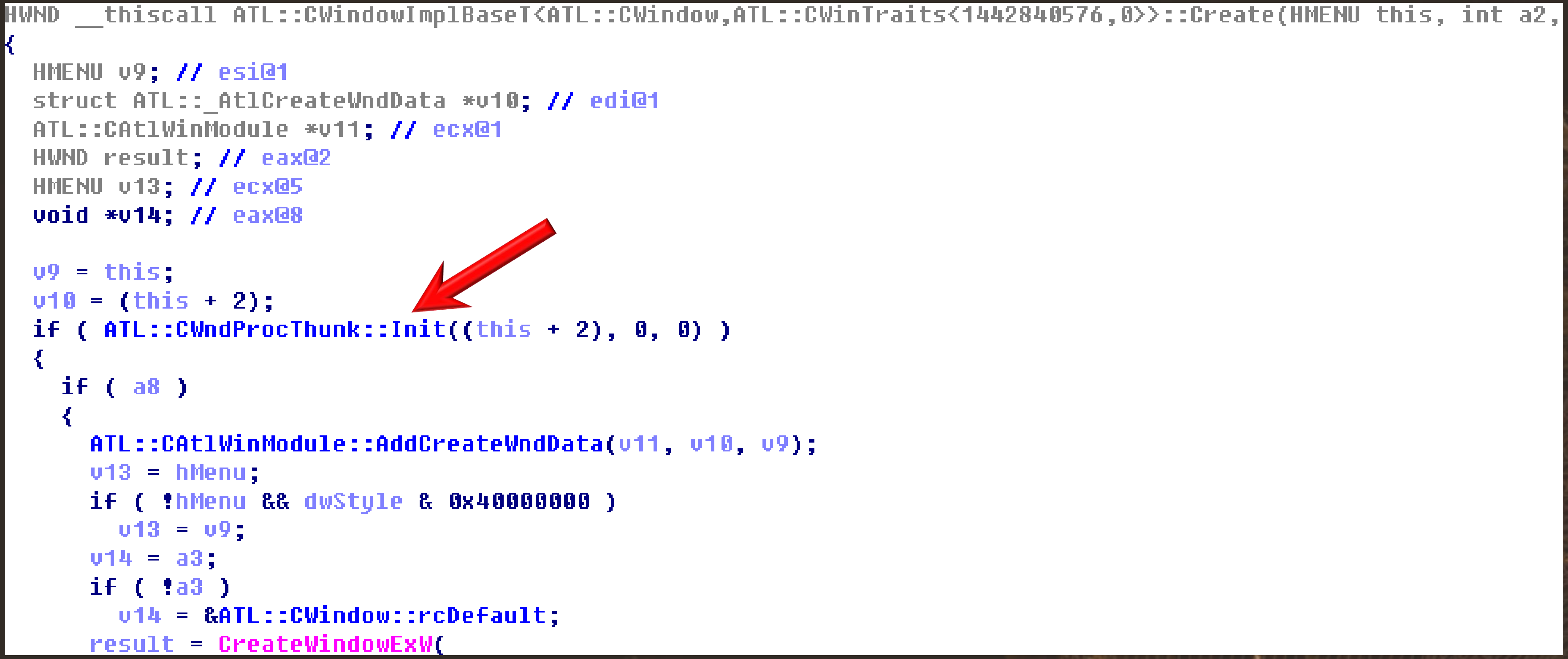
- 微软在 2013 年启动了赏金计划
- 该计划收集到很多新颖的绕过技术
- 这些绕过技术如今大多都已修复
- 部分修复并没有彻底解决问题

02

利用 ATL Thunk Pool

2.1 原始问题

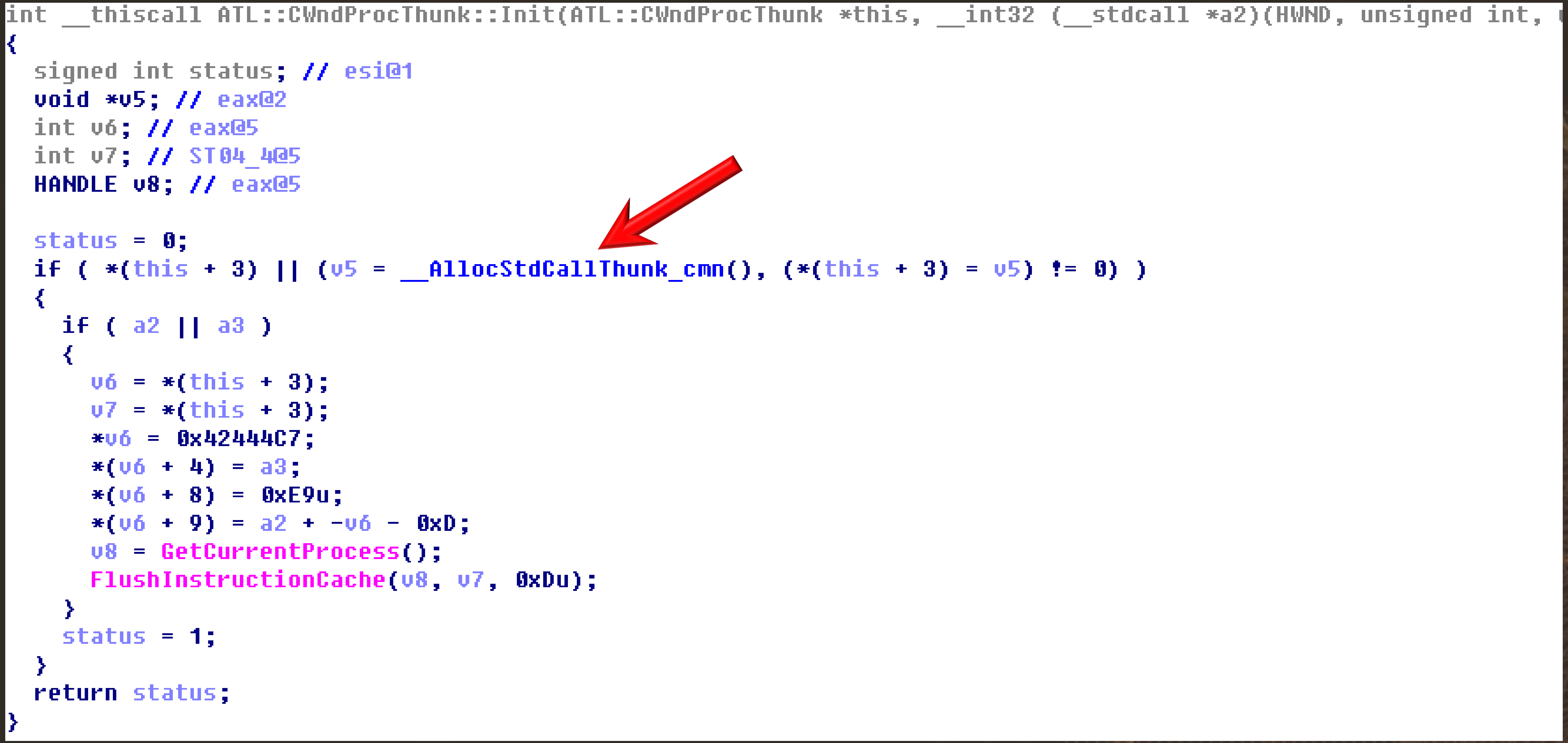
- **ATL 在创建窗口时会分配 Thunk Pool**



```
HWND __thiscall ATL::CWindowImplBaseT<ATL::CWindow,ATL::CWinTraits<1442840576,0>>::Create(HMENU this, int a2,
{
  HMENU v9; // esi@1
  struct ATL::_AtlCreateWndData *v10; // edi@1
  ATL::CAtlWinModule *v11; // ecx@1
  HWND result; // eax@2
  HMENU v13; // ecx@5
  void *v14; // eax@8

  v9 = this;
  v10 = (this + 2);
  if ( ATL::CWndProcThunk::Init((this + 2), 0, 0) )
  {
    if ( a8 )
    {
      ATL::CAtlWinModule::AddCreateWndData(v11, v10, v9);
      v13 = hMenu;
      if ( !hMenu && dwStyle & 0x40000000 )
        v13 = v9;
      v14 = a3;
      if ( !a3 )
        v14 = &ATL::CWindow::rcDefault;
      result = CreateWindowExW(
```

2.1 原始问题

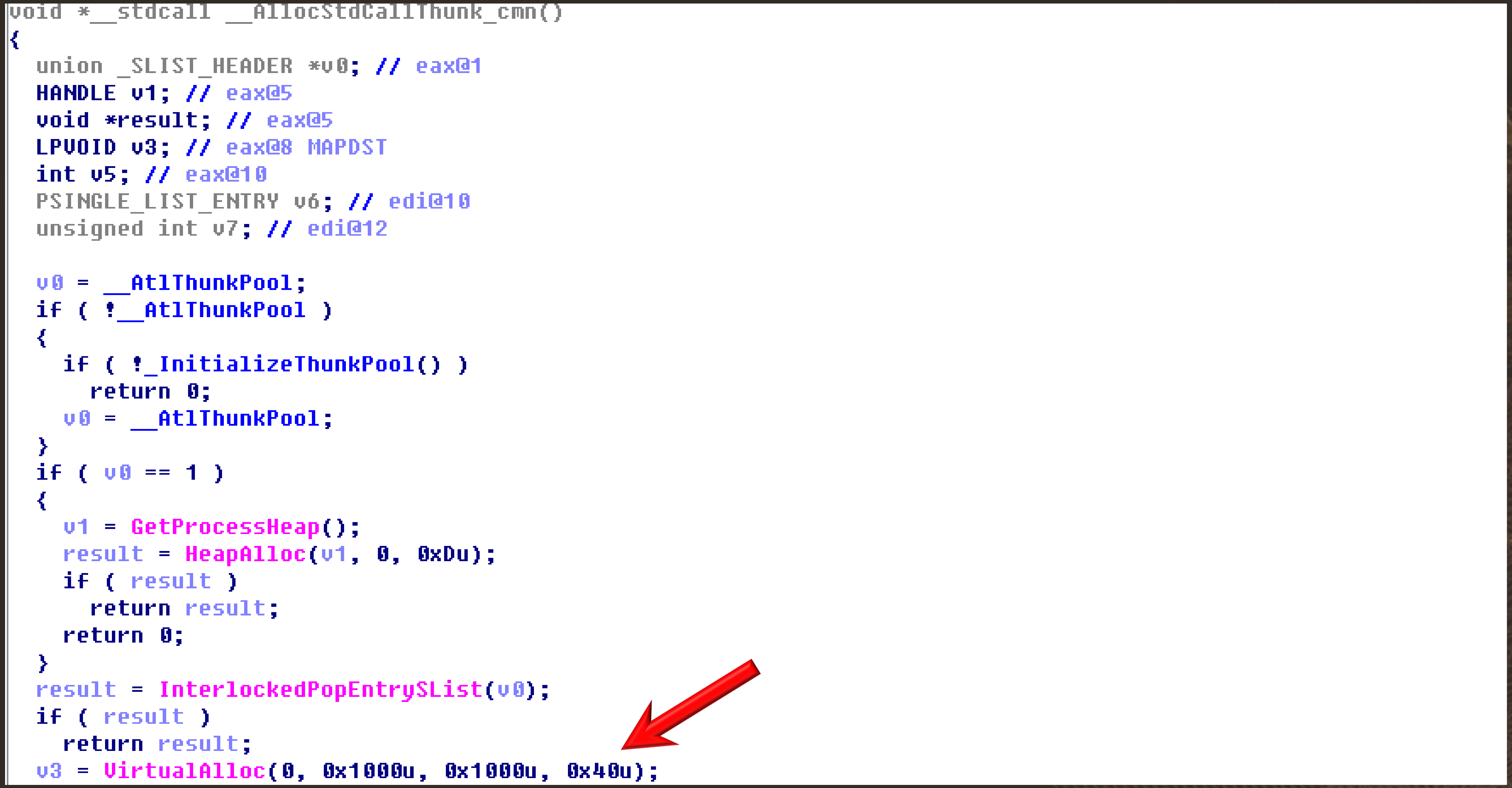
- **ATL 在创建窗口时会分配 Thunk Pool**



```
int __thiscall ATL::CWndProcThunk::Init(ATL::CWndProcThunk *this, __int32 (__stdcall *a2)(HWND, unsigned int,
{
  signed int status; // esi@1
  void *v5; // eax@2
  int v6; // eax@5
  int v7; // ST04_4@5
  HANDLE v8; // eax@5

  status = 0;
  if ( *(this + 3) || (v5 = __AllocStdCallThunk_cmn(), (*(this + 3) = v5) != 0) )
  {
    if ( a2 || a3 )
    {
      v6 = *(this + 3);
      v7 = *(this + 3);
      *v6 = 0x42444C7;
      *(v6 + 4) = a3;
      *(v6 + 8) = 0xE9u;
      *(v6 + 9) = a2 + -v6 - 0xD;
      v8 = GetCurrentProcess();
      FlushInstructionCache(v8, v7, 0xDu);
    }
    status = 1;
  }
  return status;
}
```

2.1 原始问题

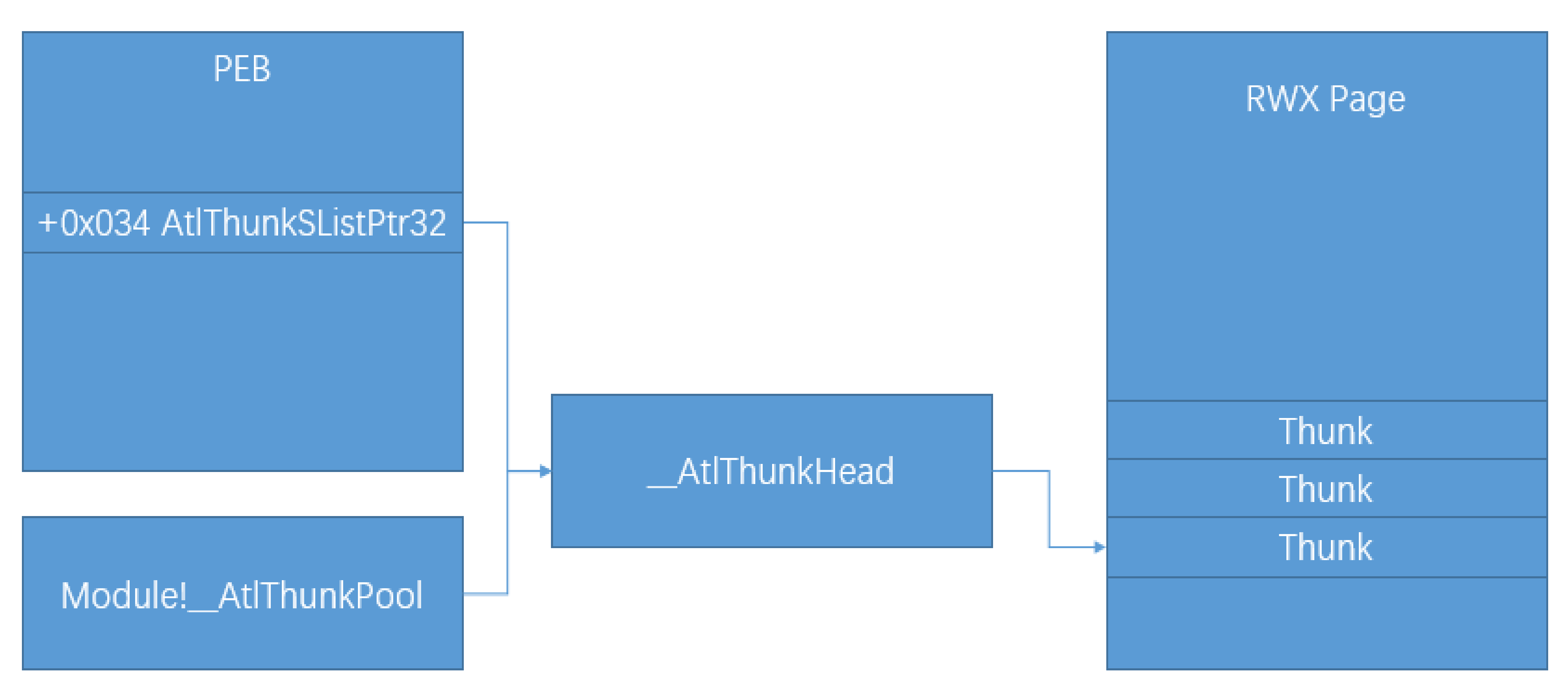
- ATL Thunk Pool 是可读、可写、可执行的页面



```
void *__stdcall __AllocStdCallThunk_cmn()
{
  union _SLIST_HEADER *v0; // eax@1
  HANDLE v1; // eax@5
  void *result; // eax@5
  LPVOID v3; // eax@8 MAPDST
  int v5; // eax@10
  PSINGLE_LIST_ENTRY v6; // edi@10
  unsigned int v7; // edi@12

  v0 = __AtlThunkPool;
  if ( !__AtlThunkPool )
  {
    if ( !_InitializeThunkPool() )
      return 0;
    v0 = __AtlThunkPool;
  }
  if ( v0 == 1 )
  {
    v1 = GetProcessHeap();
    result = HeapAlloc(v1, 0, 0xDu);
    if ( result )
      return result;
    return 0;
  }
  result = InterlockedPopEntrySList(v0);
  if ( result )
    return result;
  v3 = VirtualAlloc(0, 0x1000u, 0x1000u, 0x40u);
```

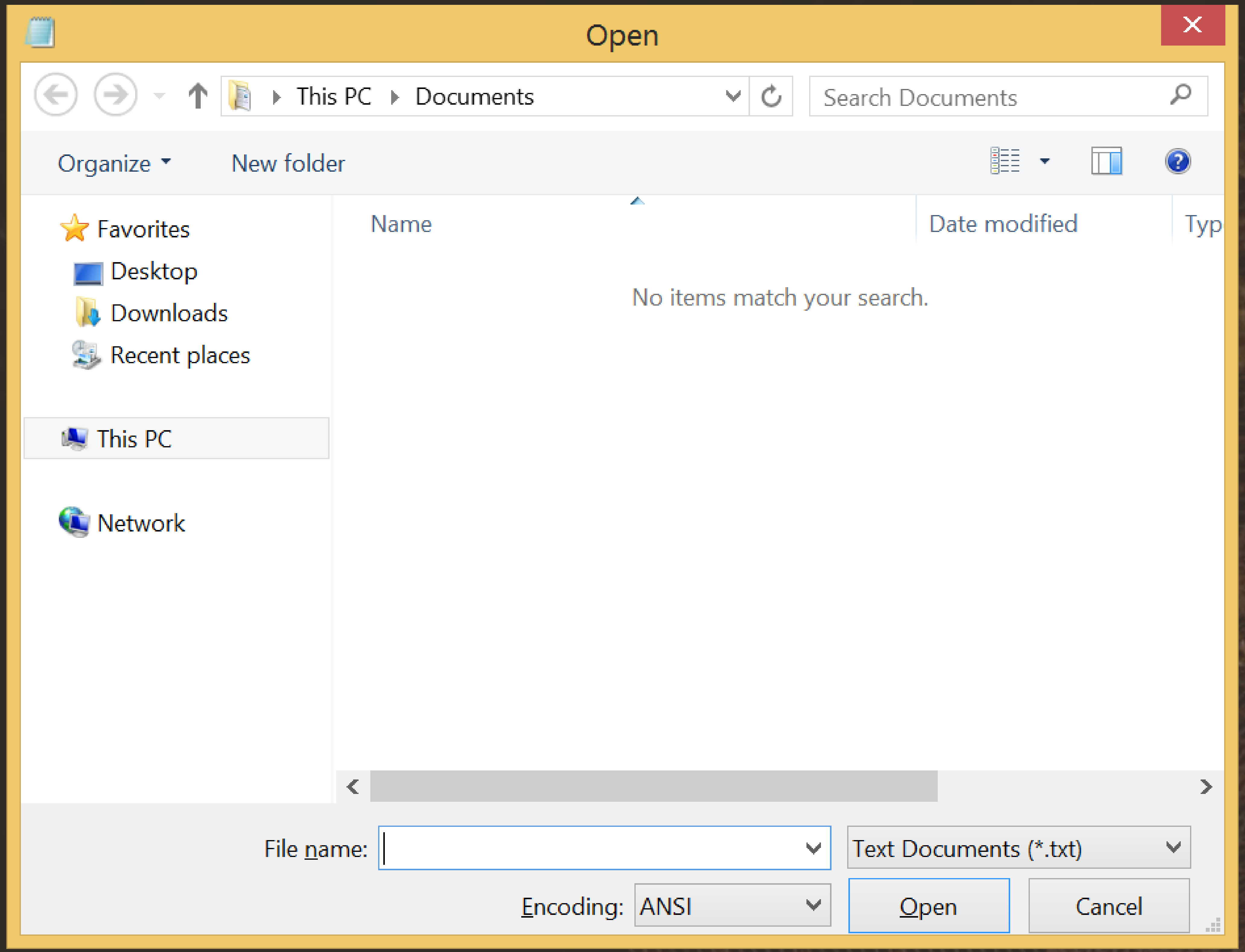
2.1 原始问题

- **获取 ATL Thunk Pool 的地址**



2.1 原始问题

- 使用 ATL Thunk Pool 的组件



No items match your search.

File name:

Text Documents (*.txt)

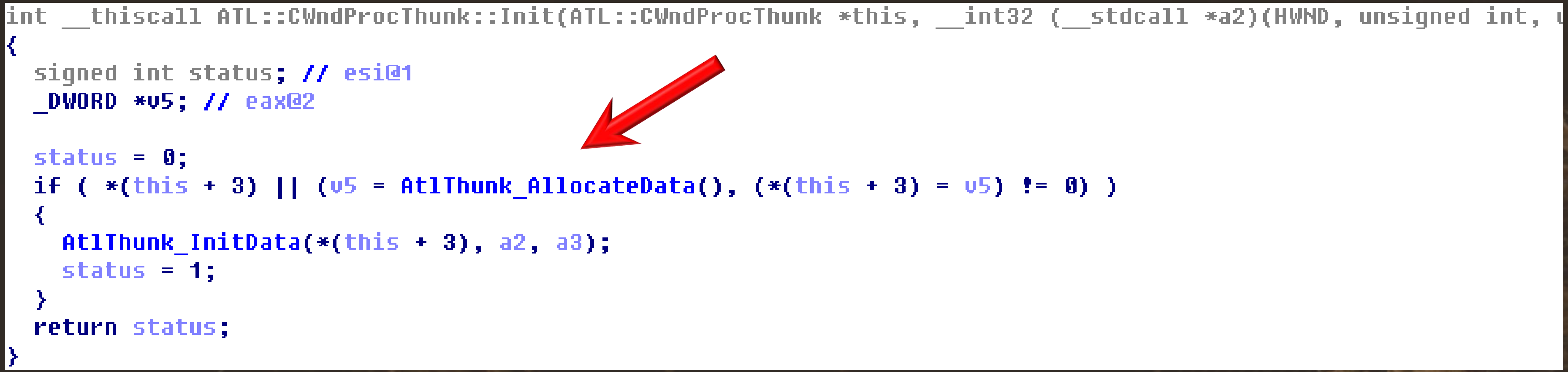
Encoding: ANSI

2.1 原始问题

- **利用方案**
  - 插入 flash 以触发 ATL Thunk Pool 的分配
  - 读取 ATL Thunk Pool 的地址
  - 向 ATL Thunk Pool 中写入 shellcode
  - 执行 shellcode

2.2 第一次修复

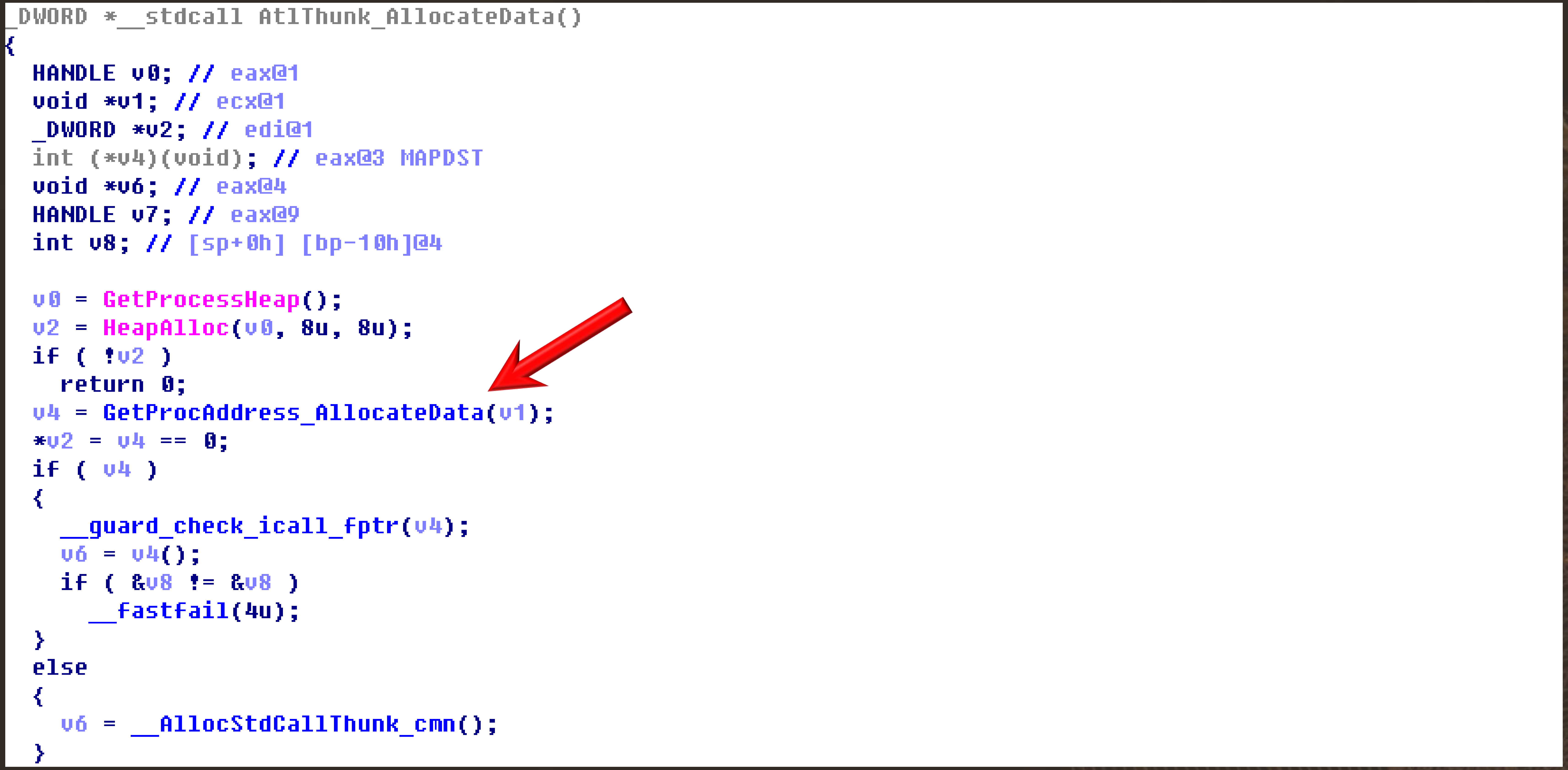
- 用 AtlThunk_AllocateData 代替 __AllocStdCallThunk_cmn



```
int __thiscall ATL::CWndProcThunk::Init(ATL::CWndProcThunk *this, __int32 (__stdcall *a2)(HWND, unsigned int, u
{
  signed int status; // esi@1
  _DWORD *v5; // eax@2

  status = 0;
  if ( *(this + 3) || (v5 = AtlThunk_AllocateData(), (*(this + 3) = v5) != 0) )
  {
    AtlThunk_InitData(*(this + 3), a2, a3);
    status = 1;
  }
  return status;
}
```

2.2 第一次修复

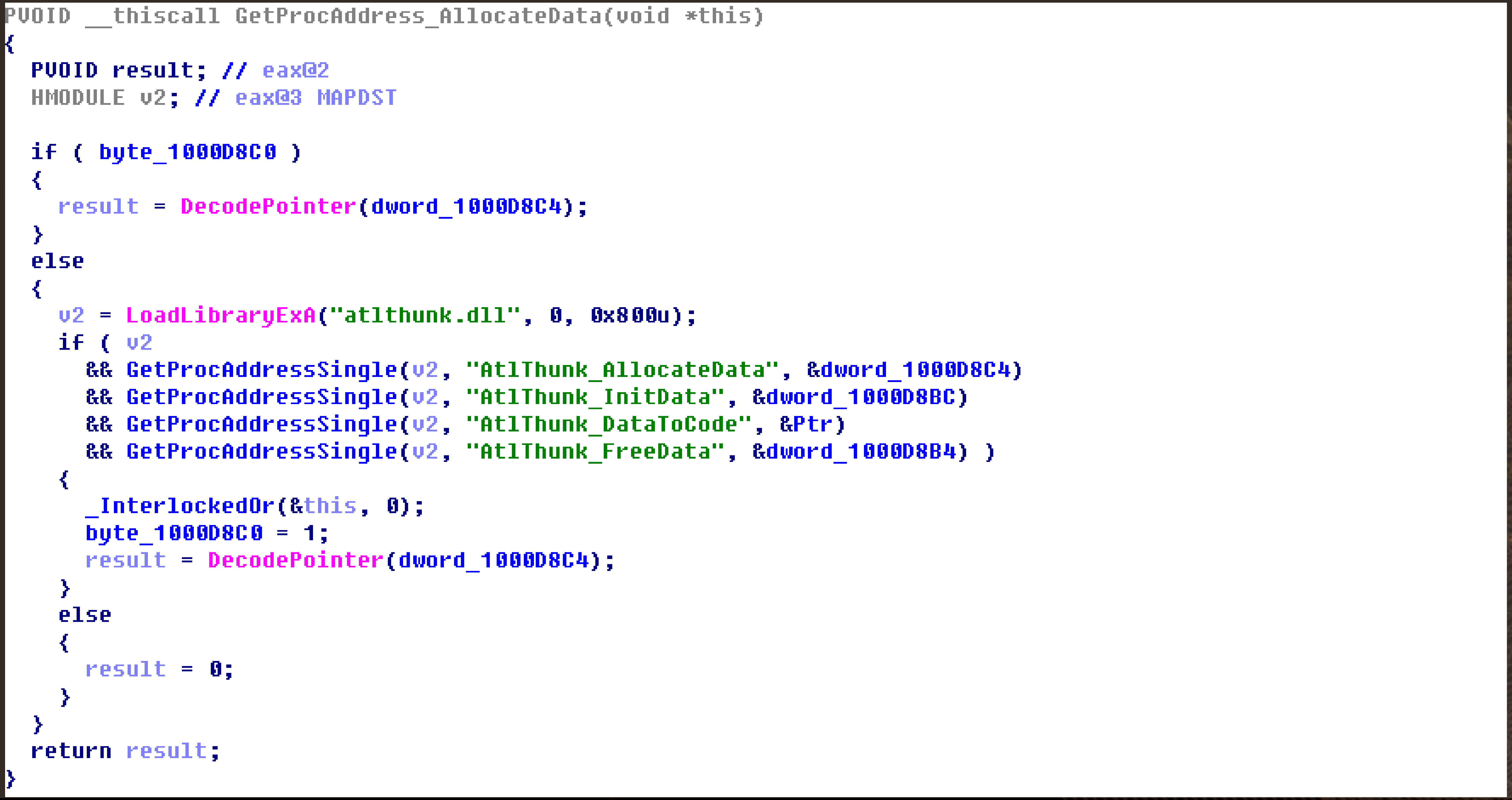
- **AtlThunk_AllocateData 在 atlthunk.dll 中实现**



```
_DWORD *__stdcall AtlThunk_AllocateData()
{
  HANDLE v0; // eax@1
  void *v1; // ecx@1
  _DWORD *v2; // edi@1
  int (*v4)(void); // eax@3 MAPDST
  void *v6; // eax@4
  HANDLE v7; // eax@9
  int v8; // [sp+0h] [bp-10h]@4

  v0 = GetProcessHeap();
  v2 = HeapAlloc(v0, 8u, 8u);
  if ( !v2 )
    return 0;
  v4 = GetProcAddress_AllocateData(v1);
  *v2 = v4 == 0;
  if ( v4 )
  {
    __guard_check_icall_fptr(v4);
    v6 = v4();
    if ( &v8 != &v8 )
      __fastfail(4u);
  }
  else
  {
    v6 = __AllocStdCallThunk_cmn();
  }
```

2.2 第一次修复

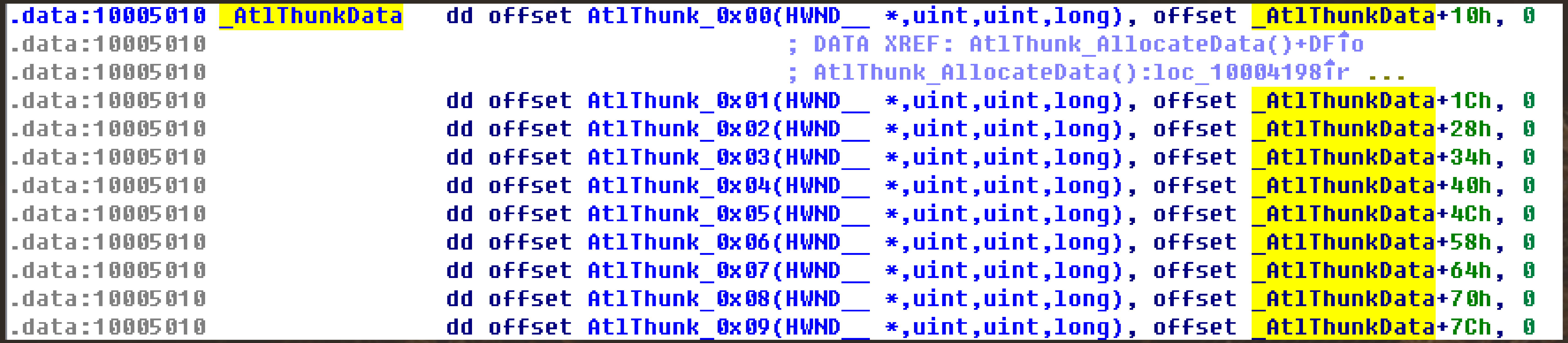
- **AtlThunk_AllocateData 在 atlthunk.dll 中实现**



```
PVOID __thiscall GetProcAddress_AllocateData(void *this)
{
  PVOID result; // eax@2
  HMODULE v2; // eax@3 MAPDST

  if ( byte_1000D8C0 )
  {
    result = DecodePointer(dword_1000D8C4);
  }
  else
  {
    v2 = LoadLibraryExA("atlthunk.dll", 0, 0x800u);
    if ( v2
      && GetProcAddressSingle(v2, "AtlThunk_AllocateData", &dword_1000D8C4)
      && GetProcAddressSingle(v2, "AtlThunk_InitData", &dword_1000D8BC)
      && GetProcAddressSingle(v2, "AtlThunk_DataToCode", &Ptr)
      && GetProcAddressSingle(v2, "AtlThunk_FreeData", &dword_1000D8B4) )
    {
      _InterlockedOr(&this, 0);
      byte_1000D8C0 = 1;
      result = DecodePointer(dword_1000D8C4);
    }
    else
    {
      result = 0;
    }
  }
  return result;
}
```

2.2 第一次修复

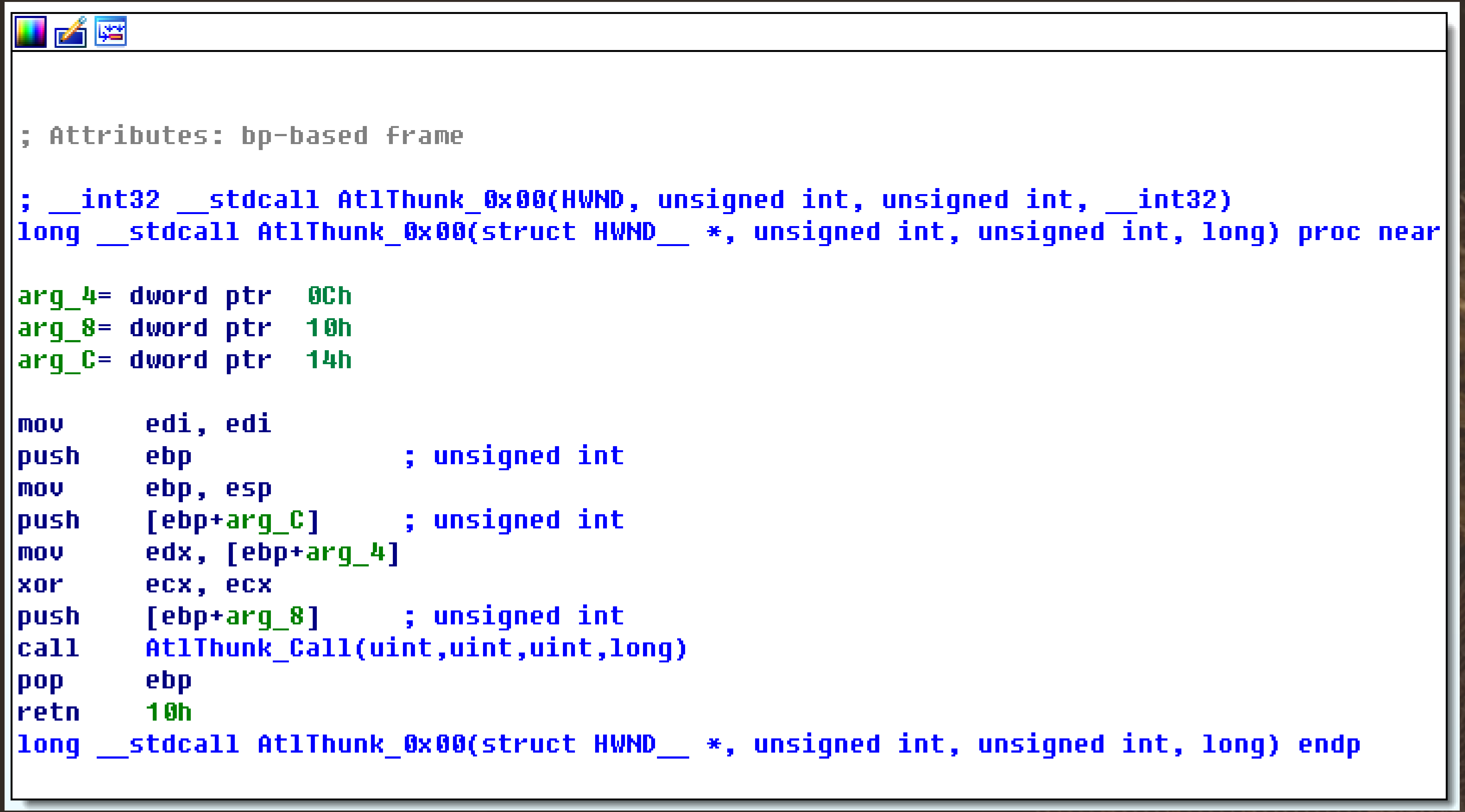
- atlthunk.dll 将数据与代码分离



```
.data:10005010 _AtlThunkData   dd offset AtlThunk_0x00(HWND__ *,uint,uint,long), offset _AtlThunkData+10h, 0
.data:10005010                                         ; DATA XREF: AtlThunk_AllocateData()+DF↑o
.data:10005010                                         ; AtlThunk_AllocateData():loc_10004198↑r ...
.data:10005010                 dd offset AtlThunk_0x01(HWND__ *,uint,uint,long), offset _AtlThunkData+1Ch, 0
.data:10005010                 dd offset AtlThunk_0x02(HWND__ *,uint,uint,long), offset _AtlThunkData+28h, 0
.data:10005010                 dd offset AtlThunk_0x03(HWND__ *,uint,uint,long), offset _AtlThunkData+34h, 0
.data:10005010                 dd offset AtlThunk_0x04(HWND__ *,uint,uint,long), offset _AtlThunkData+40h, 0
.data:10005010                 dd offset AtlThunk_0x05(HWND__ *,uint,uint,long), offset _AtlThunkData+4Ch, 0
.data:10005010                 dd offset AtlThunk_0x06(HWND__ *,uint,uint,long), offset _AtlThunkData+58h, 0
.data:10005010                 dd offset AtlThunk_0x07(HWND__ *,uint,uint,long), offset _AtlThunkData+64h, 0
.data:10005010                 dd offset AtlThunk_0x08(HWND__ *,uint,uint,long), offset _AtlThunkData+70h, 0
.data:10005010                 dd offset AtlThunk_0x09(HWND__ *,uint,uint,long), offset _AtlThunkData+7Ch, 0
```

2.2 第一次修复

- atlthunk.dll 将数据与代码分离



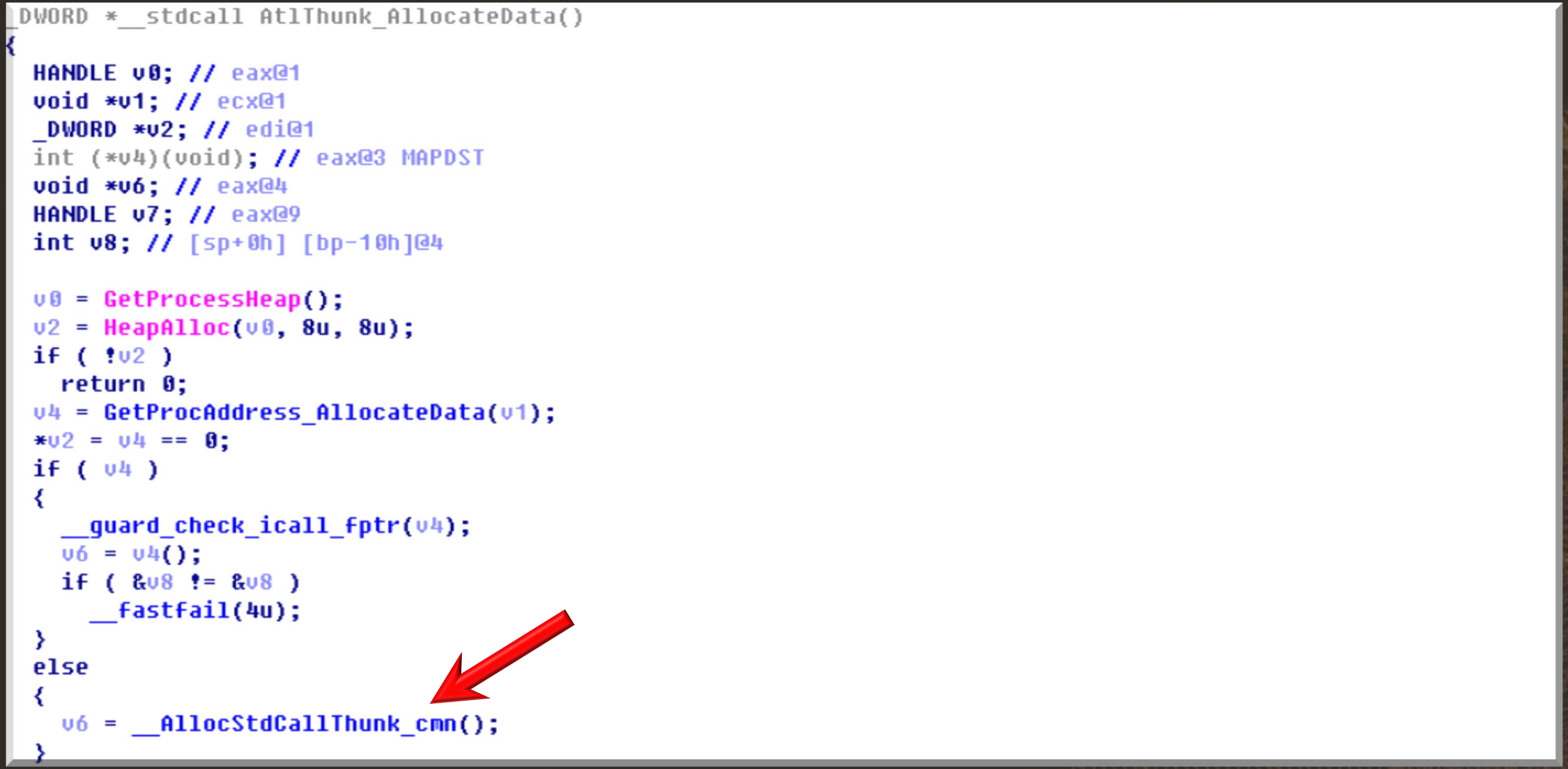
```
; Attributes: bp-based frame

; __int32 __stdcall AtlThunk_0x00(HWND, unsigned int, unsigned int, __int32)
long __stdcall AtlThunk_0x00(struct HWND__ *, unsigned int, unsigned int, long) proc near

arg_4= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h

mov     edi, edi
push    ebp              ; unsigned int
mov     ebp, esp
push    [ebp+arg_C]      ; unsigned int
mov     edx, [ebp+arg_4]
xor     ecx, ecx
push    [ebp+arg_8]      ; unsigned int
call    AtlThunk_Call(uint,uint,uint,long)
pop     ebp
retn    10h
long __stdcall AtlThunk_0x00(struct HWND__ *, unsigned int, unsigned int, long) endp
```

2.2 第一次修复

- atlthunk.dll 将数据与代码分离
  - 数据部分可读、可写（PAGE_READWRITE）
  - 代码部分仅可执行（PAGE_EXECUTE）
  - 不再使用可读、可写、可执行（PAGE_EXECUTE_READWRITE）的页面

2.3 绕过第一次修复

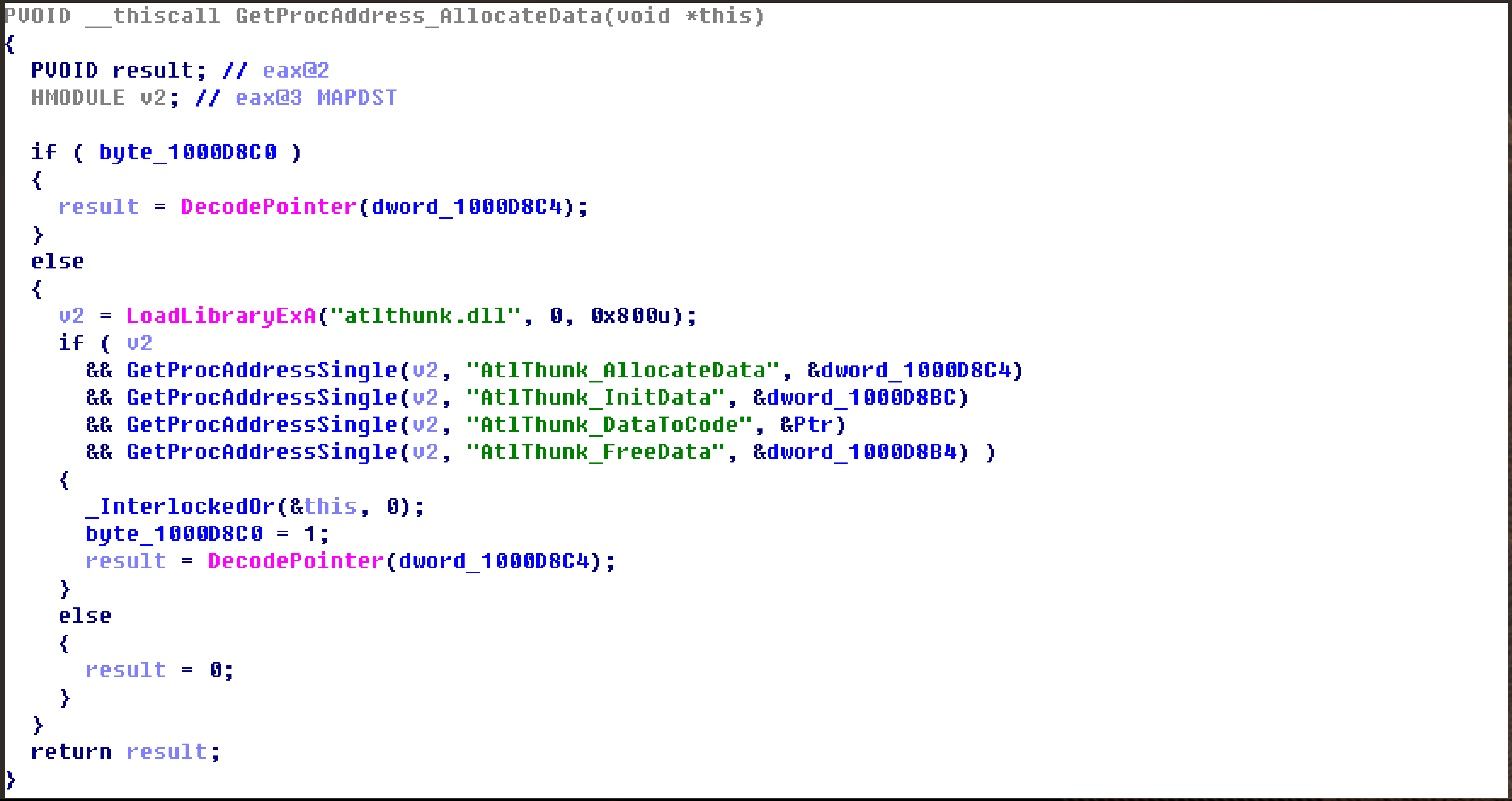
- 出于兼容性的考虑修复方案中有一个回退机制



```
DWORD *__stdcall AtlThunk_AllocateData()
{
  HANDLE v0; // eax@1
  void *v1; // ecx@1
  _DWORD *v2; // edi@1
  int (*v4)(void); // eax@3 MAPDST
  void *v6; // eax@4
  HANDLE v7; // eax@9
  int v8; // [sp+0h] [bp-10h]@4

  v0 = GetProcessHeap();
  v2 = HeapAlloc(v0, 8u, 8u);
  if ( !v2 )
    return 0;
  v4 = GetProcAddress_AllocateData(v1);
  *v2 = v4 == 0;
  if ( v4 )
  {
    __guard_check_icall_fptr(v4);
    v6 = v4();
    if ( &v8 != &v8 )
      __fastfail(4u);
  }
  else
  {
    v6 = __AllocStdCallThunk_cmn();
  }
```

2.3 绕过第一次修复

- 导致 GetProcAddress_AllocateData 失败的情况



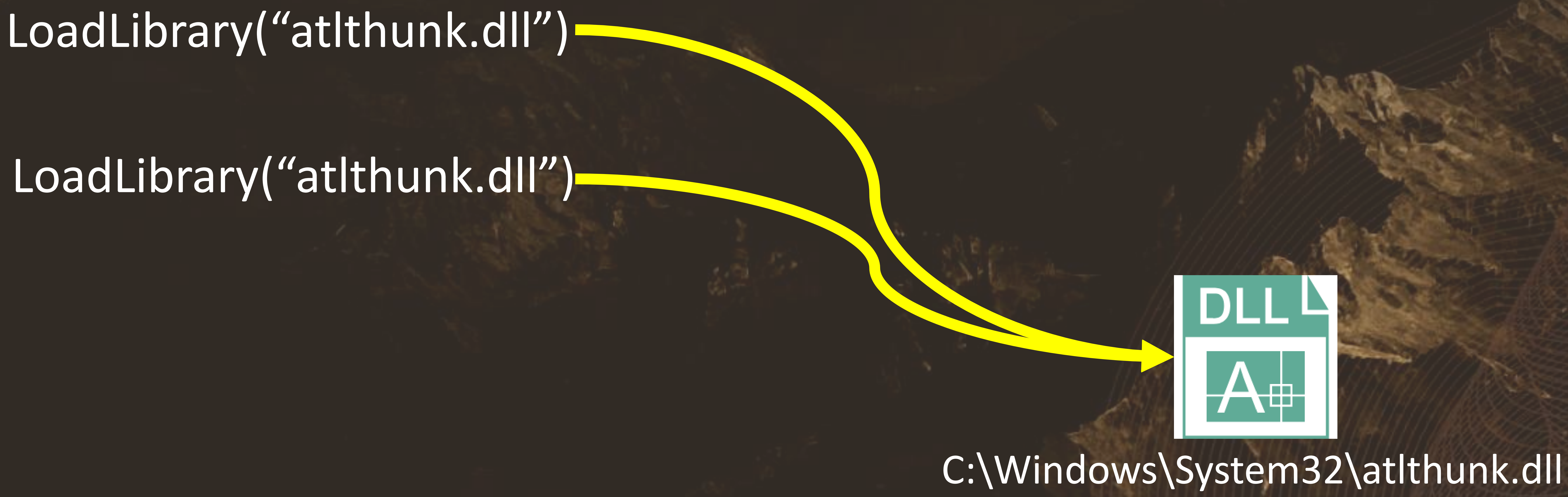
```
PVOID __thiscall GetProcAddress_AllocateData(void *this)
{
  PVOID result; // eax@2
  HMODULE v2; // eax@3 MAPDST

  if ( byte_1000D8C0 )
  {
    result = DecodePointer(dword_1000D8C4);
  }
  else
  {
    v2 = LoadLibraryExA("atlthunk.dll", 0, 0x800u);
    if ( v2
      && GetProcAddressSingle(v2, "AtlThunk_AllocateData", &dword_1000D8C4)
      && GetProcAddressSingle(v2, "AtlThunk_InitData", &dword_1000D8BC)
      && GetProcAddressSingle(v2, "AtlThunk_DataToCode", &Ptr)
      && GetProcAddressSingle(v2, "AtlThunk_FreeData", &dword_1000D8B4) )
    {
      _InterlockedOr(&this, 0);
      byte_1000D8C0 = 1;
      result = DecodePointer(dword_1000D8C4);
    }
    else
    {
      result = 0;
    }
  }
  return result;
}
```

2.3 绕过第一次修复

- 导致 GetProcAddress_AllocateData 失败的情况
  - LoadLibraryExA 失败
    - 库文件 atlthunk.dll 不存在
    - 没有应用此修复的老系统
  - GetProcAddress 失败
    - 不太可能发生

2.3 绕过第一次修复

- **LoadLibrary 再次加载的快速路径**



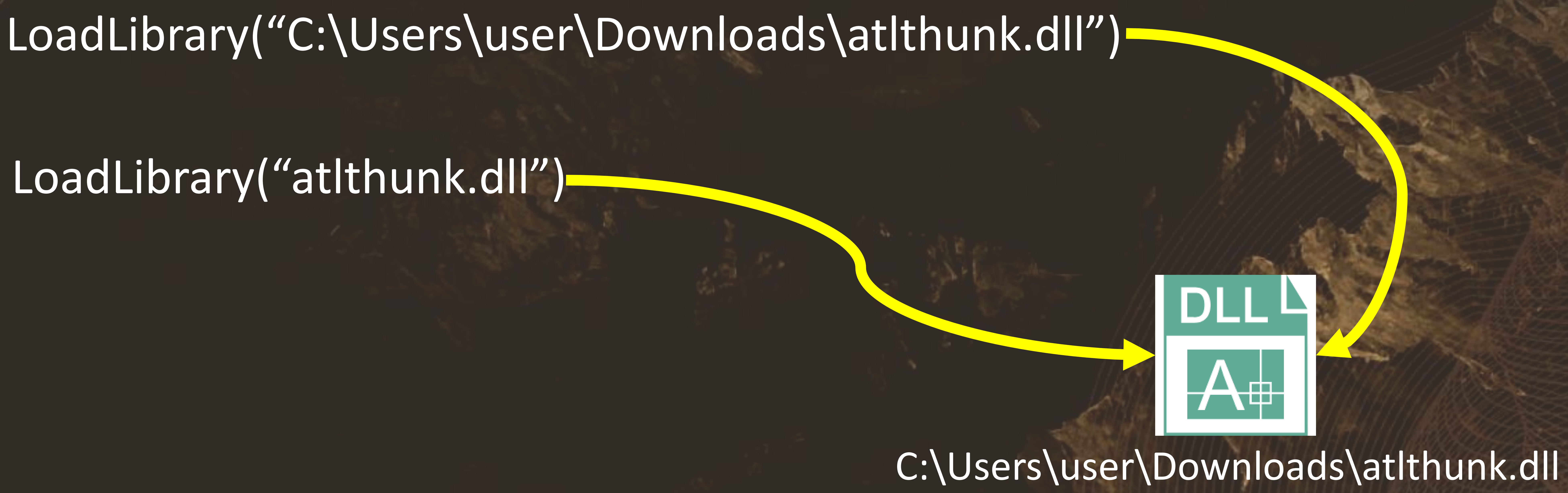
LoadLibrary(“atlthunk.dll”)

LoadLibrary(“atlthunk.dll”)

C:\Windows\System32\atlthunk.dll

2.3 绕过第一次修复

- **LoadLibrary 再次加载的快速路径**



LoadLibrary(“C:\Users\user\Downloads\atlthunk.dll”)

LoadLibrary(“atlthunk.dll”)

C:\Users\user\Downloads\atlthunk.dll

2.3 绕过第一次修复

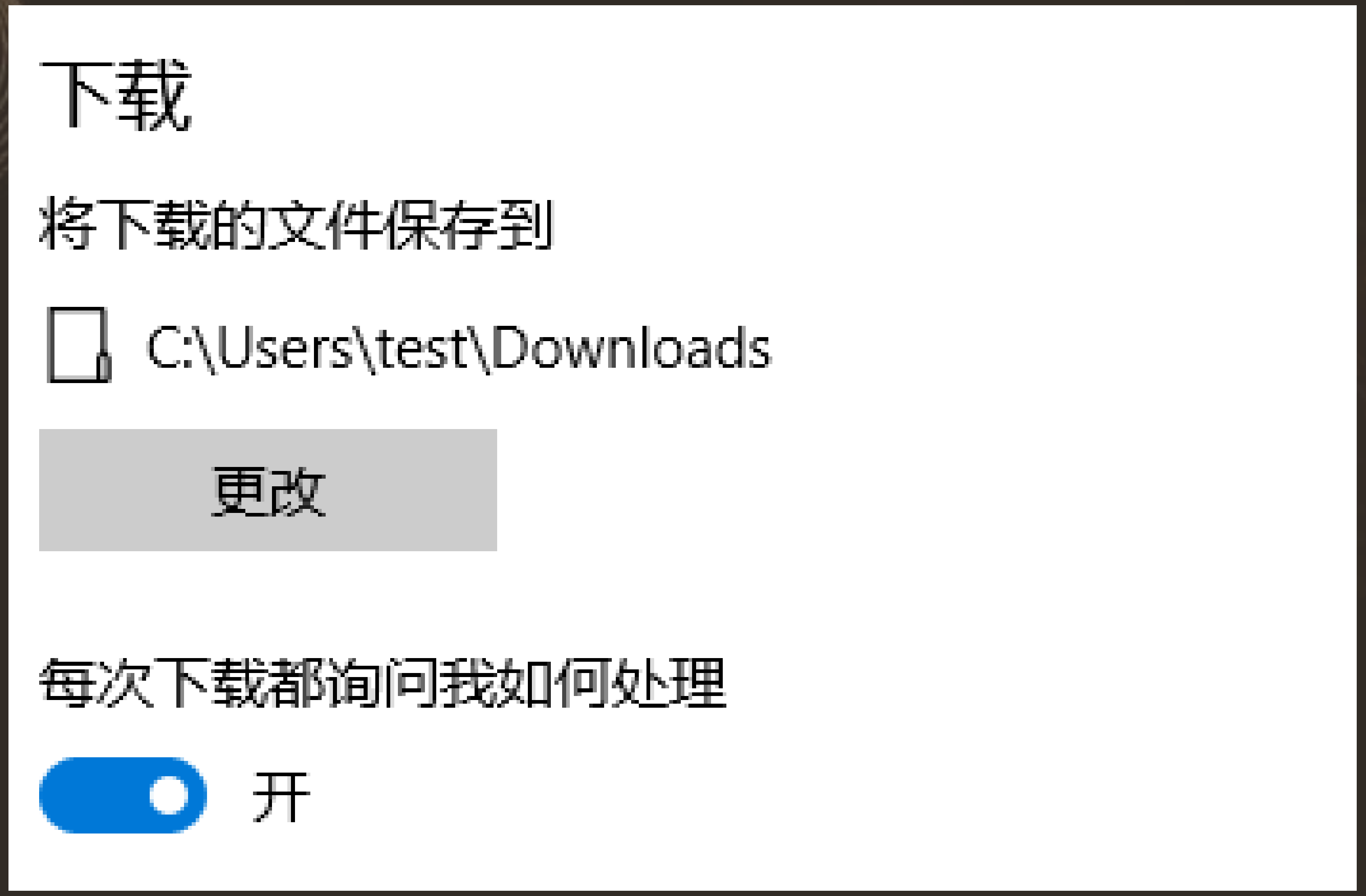
- Microsoft Edge 曾经有一个自动下载的特性
  - 访问包含 <iframe src=“path/to/atlthunk.dll”/> 的页面
  - 将自动下载 atlthunk.dll 到 %userprofile%\Downloads

2.3 绕过第一次修复

- 利用方案
  - 利用自动下载特性下载伪造的 atlthunk.dll
  - 调用 LoadLibrary 加载该 atlthunk.dll
  - 插入 flash 以触发 ATL Thunk Pool 的分配
  - 读取 ATL Thunk Pool 的地址
  - 向 ATL Thunk Pool 中写入 shellcode
  - 执行 shellcode

2.4 最终修复

- 现在 Microsoft Edge 在自动下载前会询问



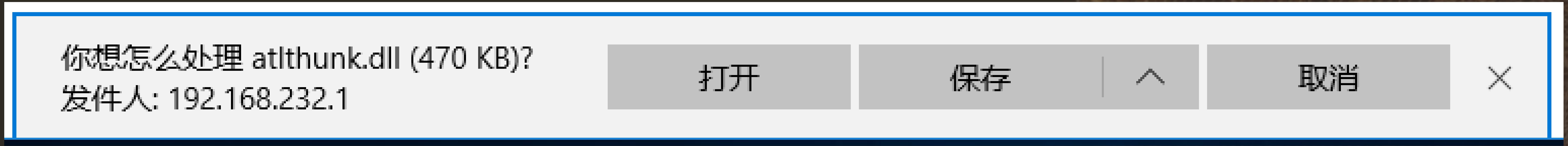
下载

将下载的文件保存到

C:\Users\test\Downloads

每次下载都询问我如何处理

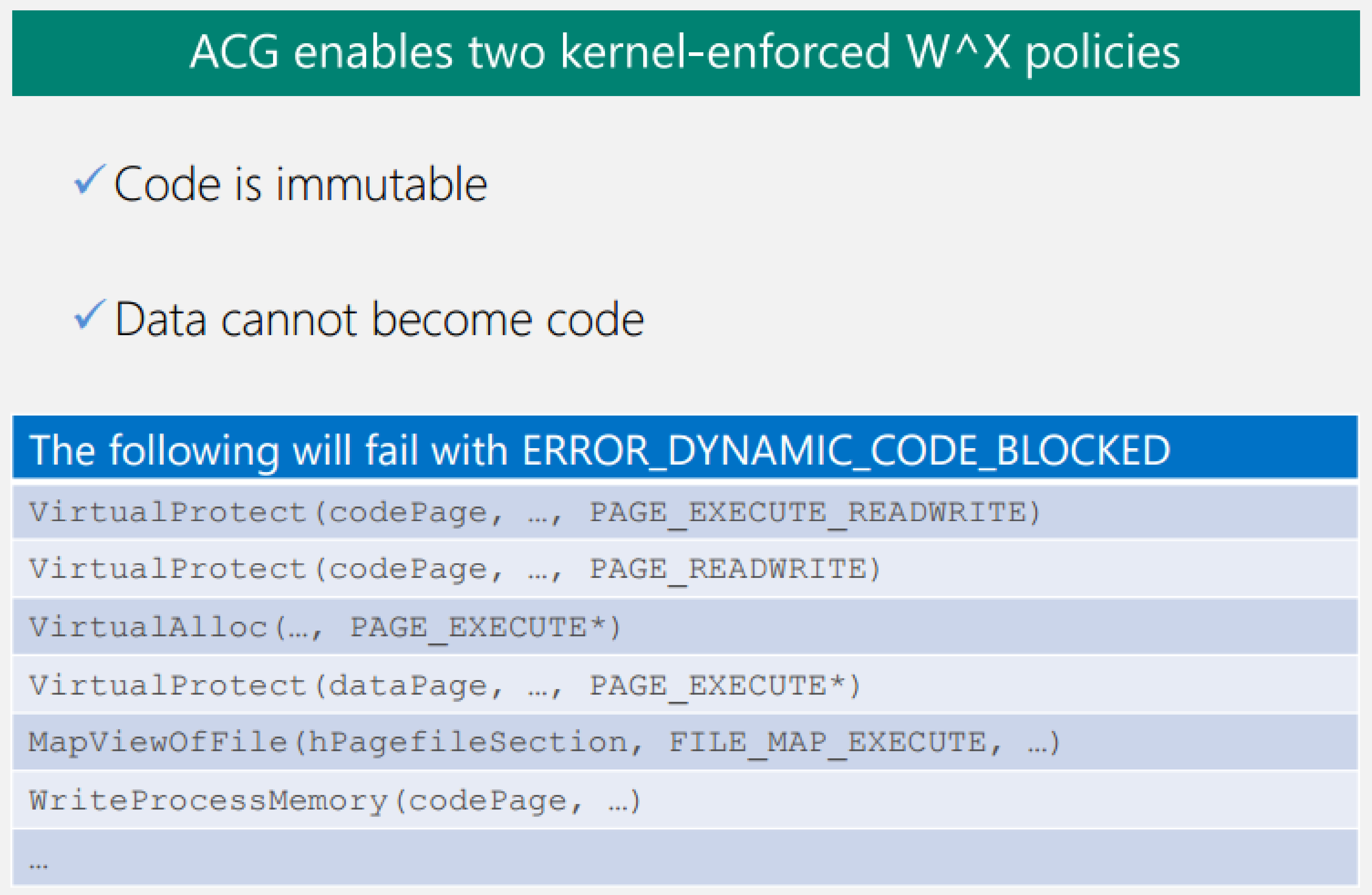
● 开



你想怎么处理 atlthunk.dll (470 KB)?
发件人: 192.168.232.1

2.4 最终修复

- ACG 阻止创建 PAGE_EXECUTE_READWRITE 页面



ACG enables two kernel-enforced W^X policies

✓ Code is immutable

✓ Data cannot become code

The following will fail with ERROR_DYNAMIC_CODE_BLOCKED

```
VirtualProtect(codePage, …, PAGE_EXECUTE_READWRITE)
VirtualProtect(codePage, …, PAGE_READWRITE)
VirtualAlloc(…, PAGE_EXECUTE*)
VirtualProtect(dataPage, …, PAGE_EXECUTE*)
MapViewOfFile(hPagefileSection, FILE_MAP_EXECUTE, …)
WriteProcessMemory(codePage, …)
…
```

03

利用 Chakra JIT Engine

3.1 原始问题

- **Chakra JIT Engine 的内存管理**

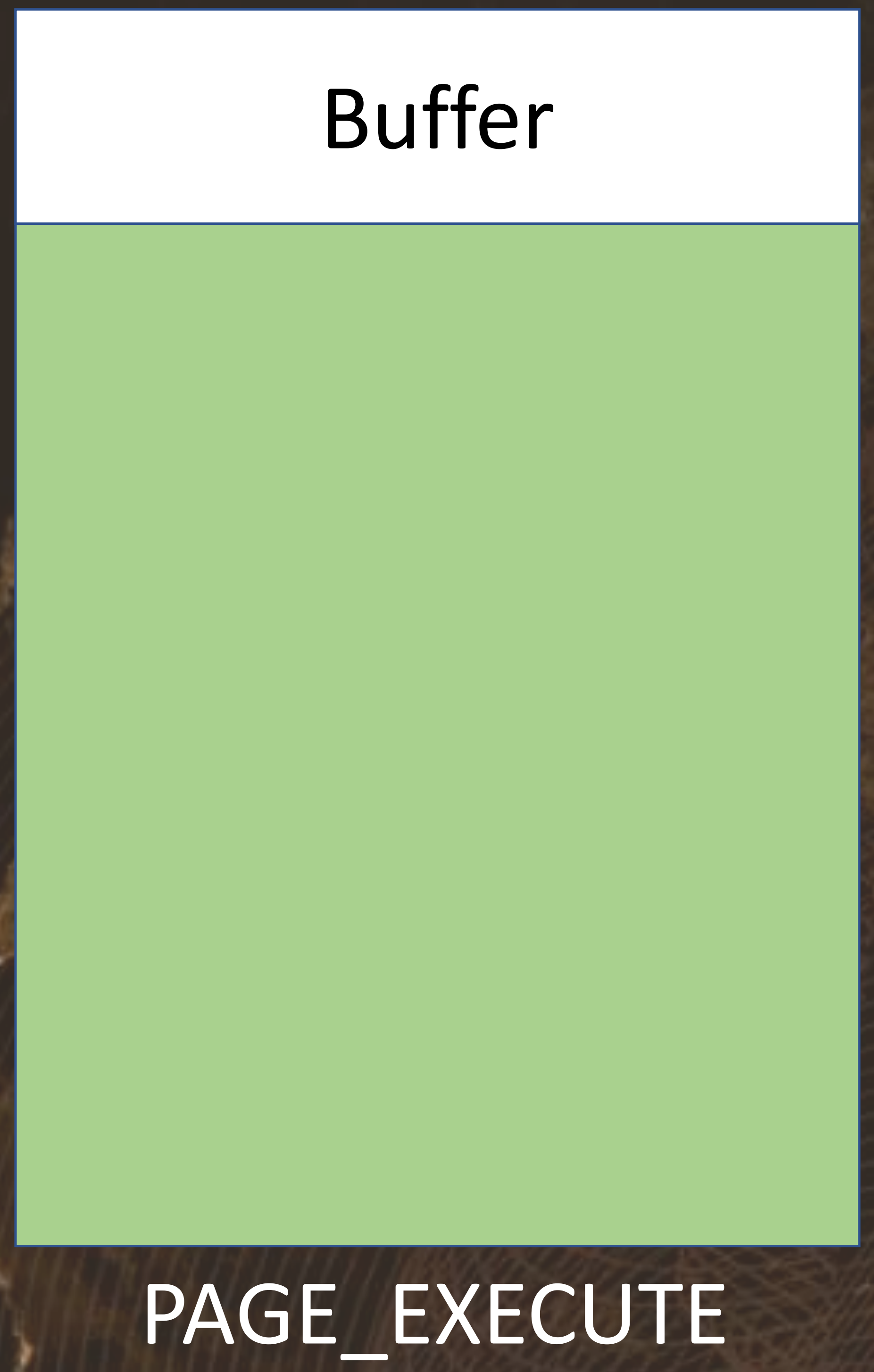
```
Encoder::Encode
    CodeGenWorkItem::RecordNativeCodeSize
        EmitBufferManager::AllocateBuffer
            EmitBufferManager::NewAllocation
                CustomHeap::Heap::Alloc
    CodeGenWorkItem::RecordNativeCode
        EmitBufferManager::CommitBuffer
            CustomHeap::Heap::ProtectAllocationWithExecuteReadWrite
            memcpy_s
            CustomHeap::Heap::ProtectAllocationWithExecuteReadOnly
```

PAGE_READWRITE

3.1 原始问题

- **Chakra JIT Engine 的内存管理**

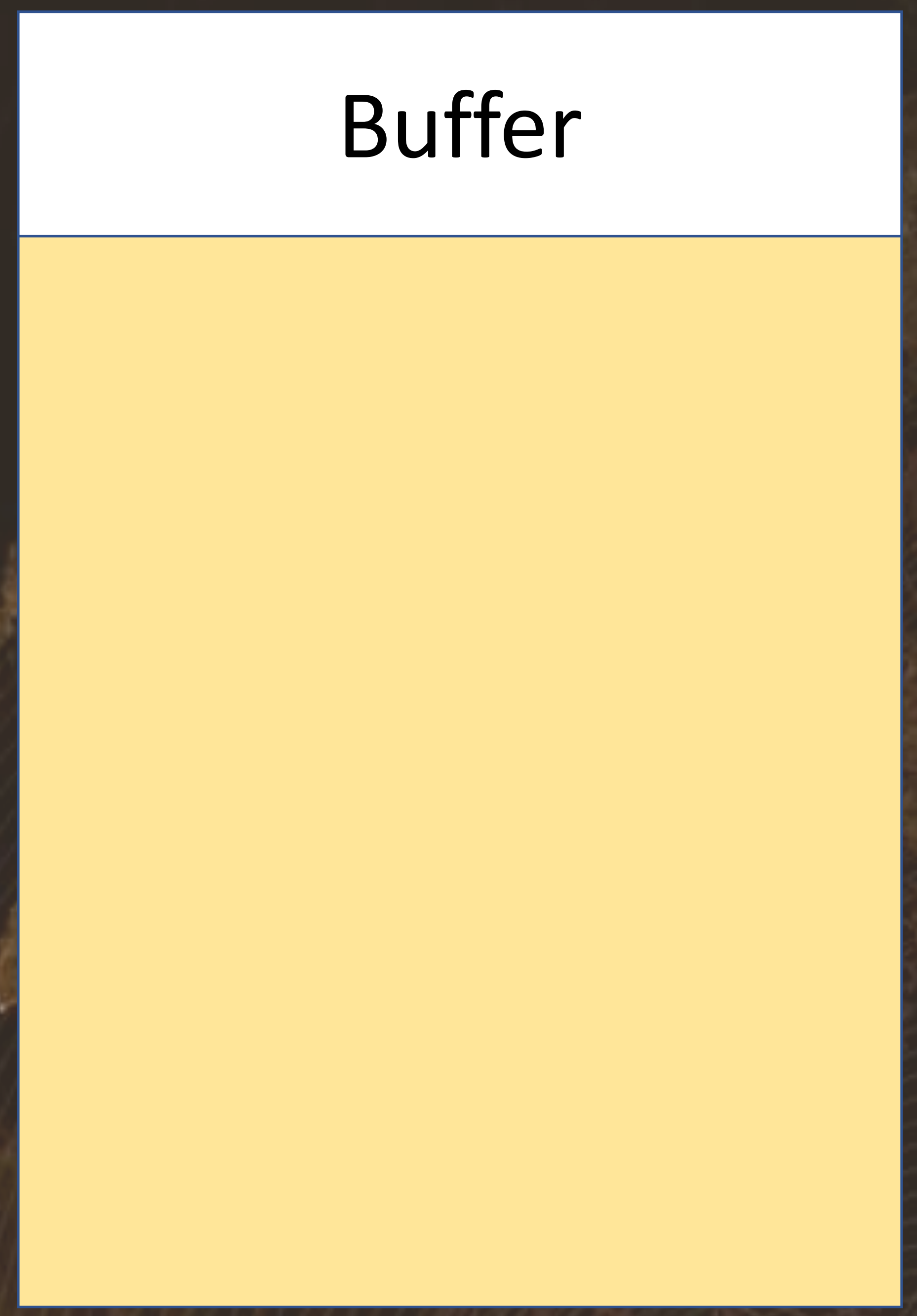
```
Encoder::Encode
    CodeGenWorkItem::RecordNativeCodeSize
        EmitBufferManager::AllocateBuffer
            EmitBufferManager::NewAllocation
                CustomHeap::Heap::Alloc
    CodeGenWorkItem::RecordNativeCode
        EmitBufferManager::CommitBuffer
            CustomHeap::Heap::ProtectAllocationWithExecuteReadWrite
            memcpy_s
            CustomHeap::Heap::ProtectAllocationWithExecuteReadOnly
```



3.1 原始问题

- **Chakra JIT Engine 的内存管理**

```
Encoder::Encode
    CodeGenWorkItem::RecordNativeCodeSize
        EmitBufferManager::AllocateBuffer
            EmitBufferManager::NewAllocation
                CustomHeap::Heap::Alloc
    CodeGenWorkItem::RecordNativeCode
        EmitBufferManager::CommitBuffer
            CustomHeap::Heap::ProtectAllocationWithExecuteReadWrite
            memcpy_s
            CustomHeap::Heap::ProtectAllocationWithExecuteReadOnly
```

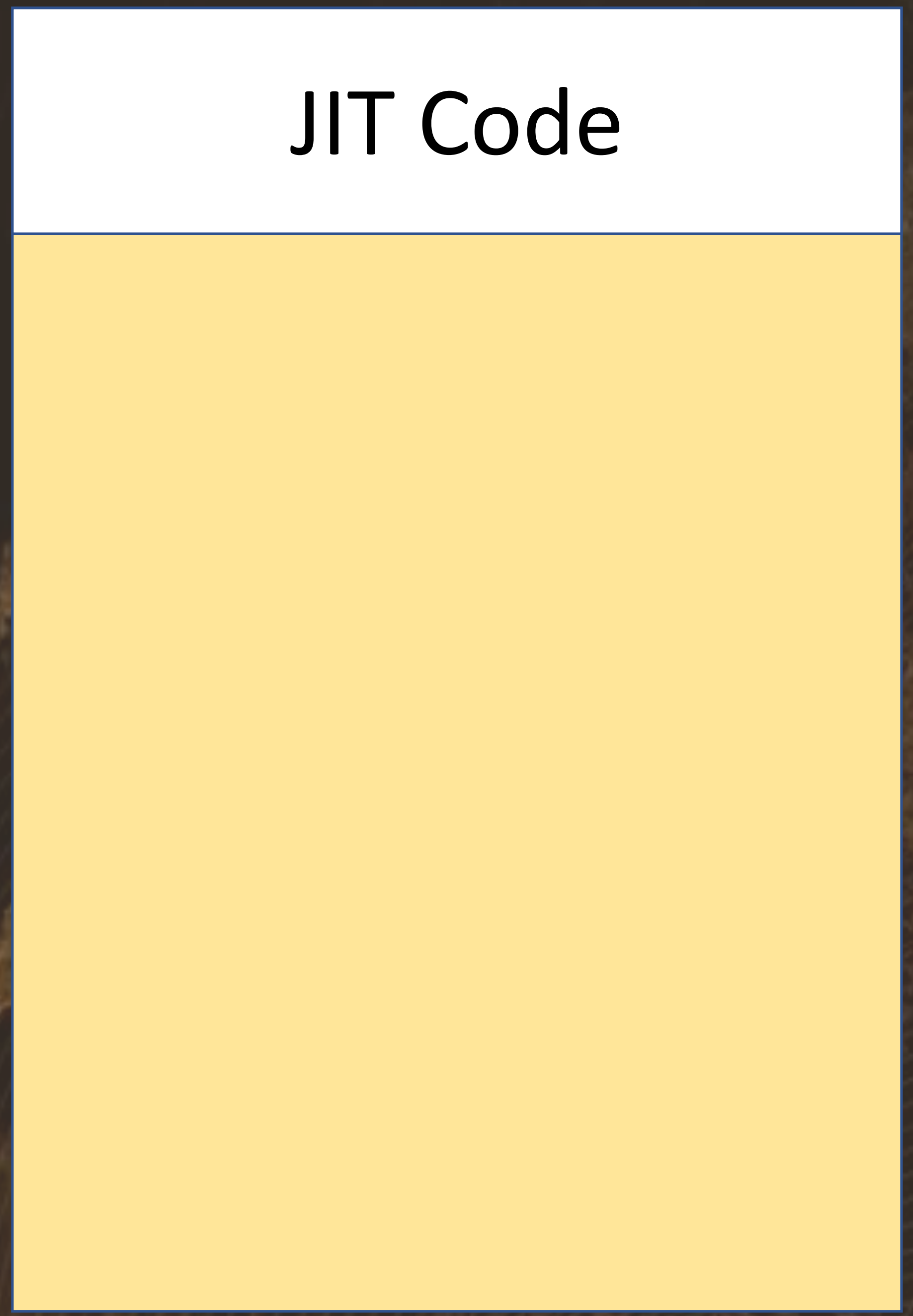


PAGE_EXECUTE_READWRITE

3.1 原始问题

- **Chakra JIT Engine 的内存管理**

```
Encoder::Encode
    CodeGenWorkItem::RecordNativeCodeSize
        EmitBufferManager::AllocateBuffer
            EmitBufferManager::NewAllocation
                CustomHeap::Heap::Alloc
    CodeGenWorkItem::RecordNativeCode
        EmitBufferManager::CommitBuffer
            CustomHeap::Heap::ProtectAllocationWithExecuteReadWrite
            memcpy_s
            CustomHeap::Heap::ProtectAllocationWithExecuteReadOnly
```

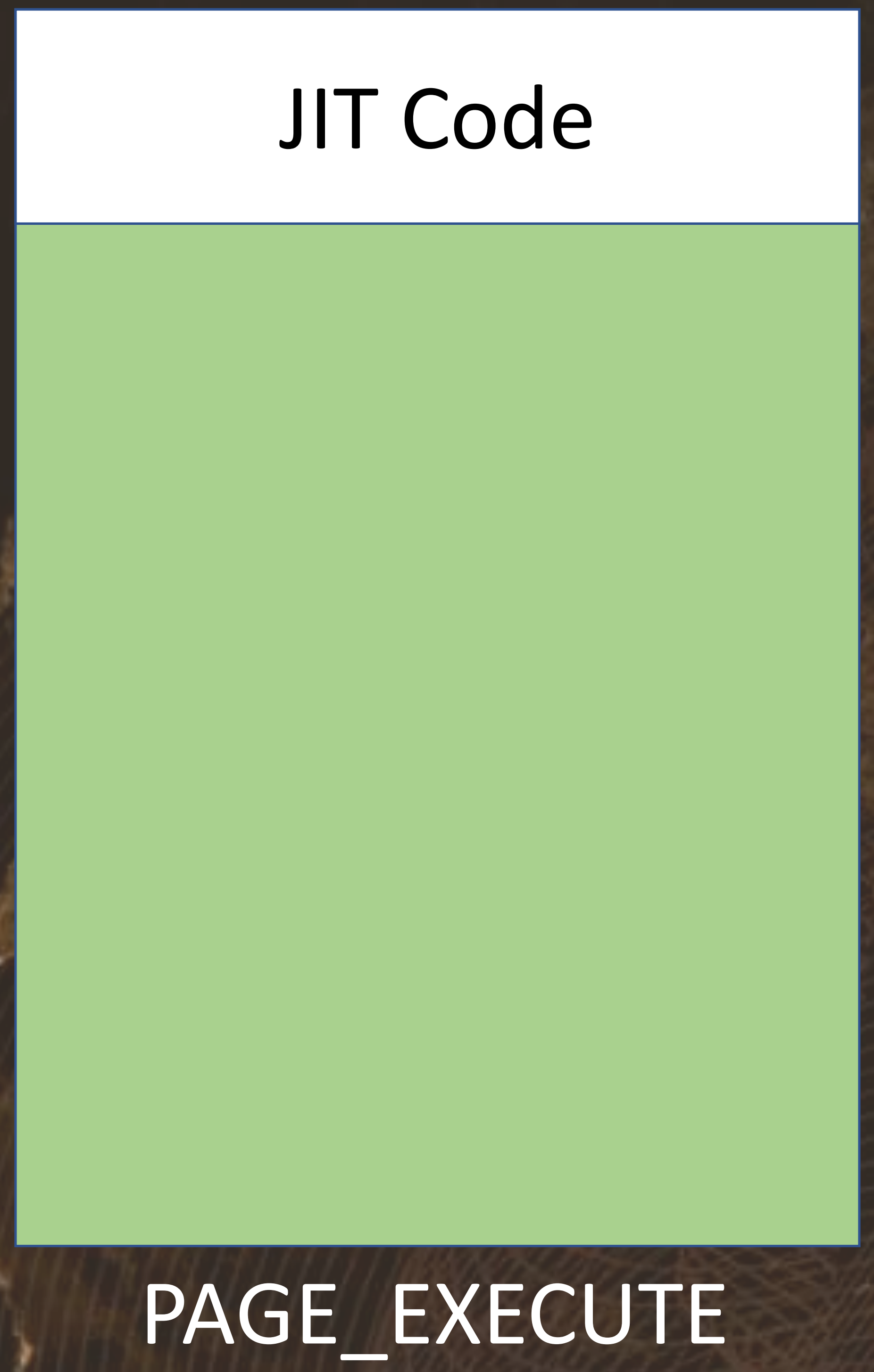


PAGE_EXECUTE_READWRITE

3.1 原始问题

- **Chakra JIT Engine 的内存管理**

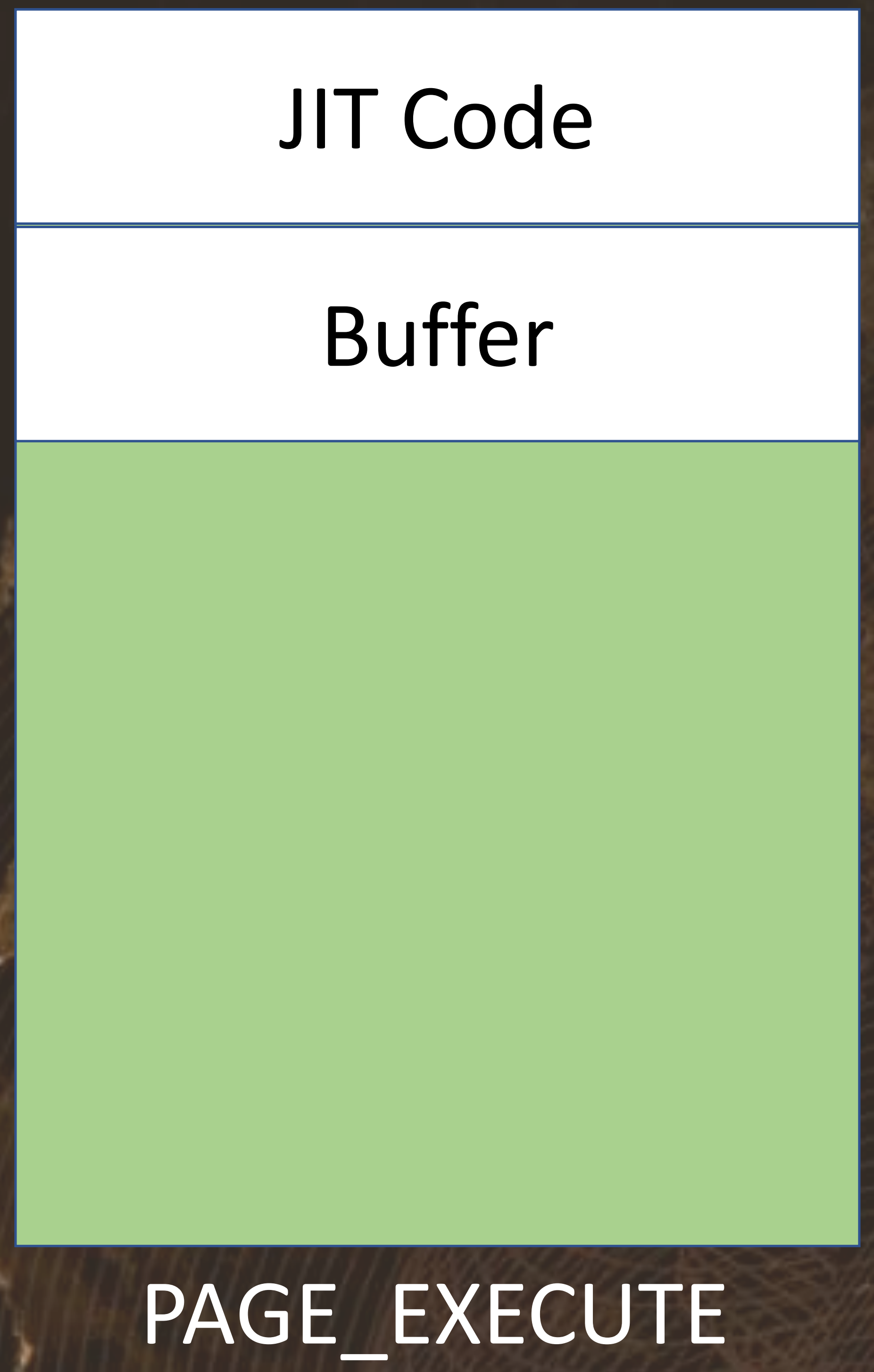
```
Encoder::Encode
    CodeGenWorkItem::RecordNativeCodeSize
        EmitBufferManager::AllocateBuffer
            EmitBufferManager::NewAllocation
                CustomHeap::Heap::Alloc
    CodeGenWorkItem::RecordNativeCode
        EmitBufferManager::CommitBuffer
            CustomHeap::Heap::ProtectAllocationWithExecuteReadWrite
            memcpy_s
            CustomHeap::Heap::ProtectAllocationWithExecuteReadOnly
```



3.1 原始问题

- **Chakra JIT Engine 的内存管理**

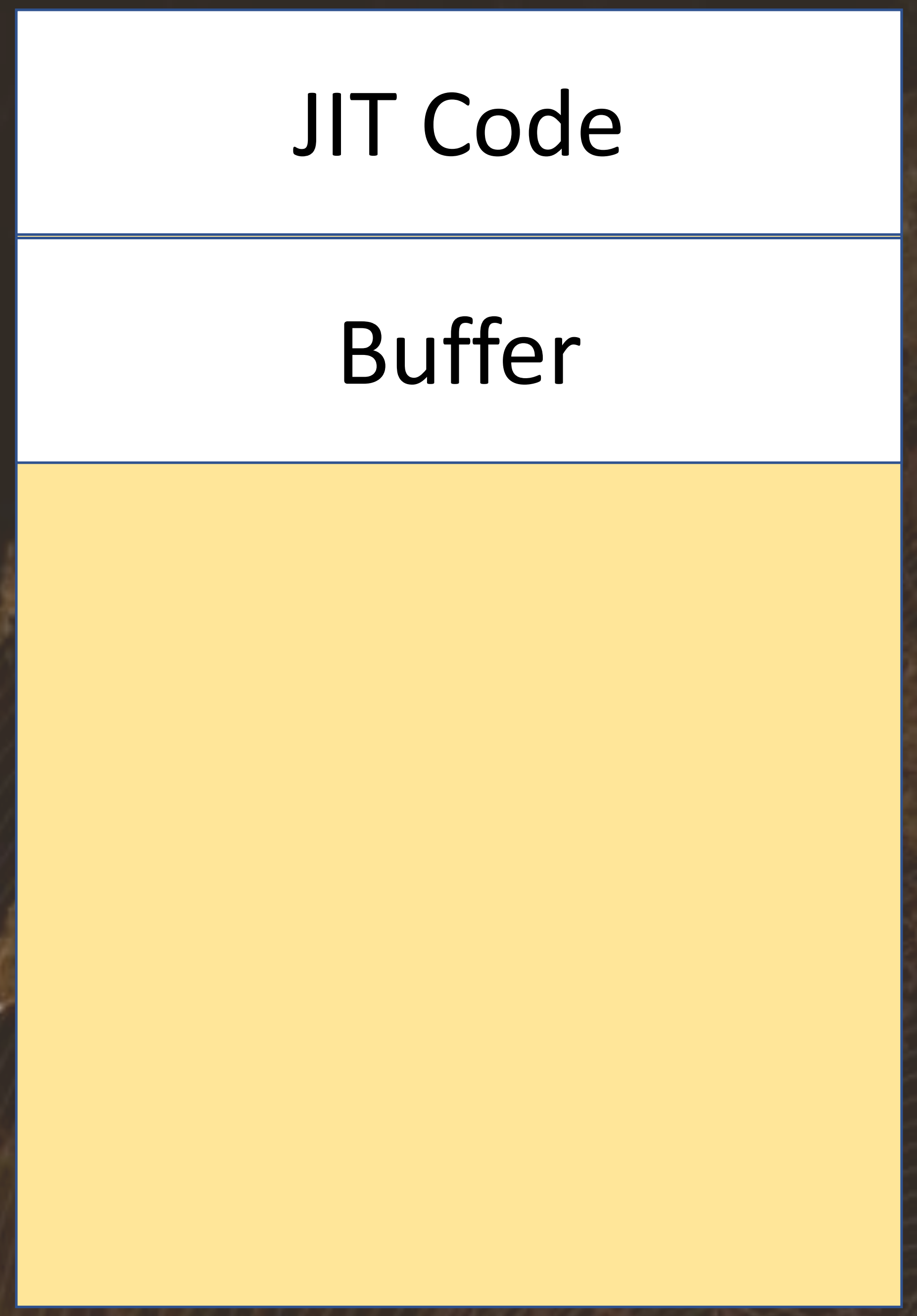
```
Encoder::Encode
    CodeGenWorkItem::RecordNativeCodeSize
        EmitBufferManager::AllocateBuffer
            EmitBufferManager::NewAllocation
                CustomHeap::Heap::Alloc
    CodeGenWorkItem::RecordNativeCode
        EmitBufferManager::CommitBuffer
            CustomHeap::Heap::ProtectAllocationWithExecuteReadWrite
            memcpy_s
            CustomHeap::Heap::ProtectAllocationWithExecuteReadOnly
```



3.1 原始问题

- **Chakra JIT Engine 的内存管理**

```
Encoder::Encode
    CodeGenWorkItem::RecordNativeCodeSize
        EmitBufferManager::AllocateBuffer
            EmitBufferManager::NewAllocation
                CustomHeap::Heap::Alloc
    CodeGenWorkItem::RecordNativeCode
        EmitBufferManager::CommitBuffer
            CustomHeap::Heap::ProtectAllocationWithExecuteReadWrite
            memcpy_s
            CustomHeap::Heap::ProtectAllocationWithExecuteReadOnly
```

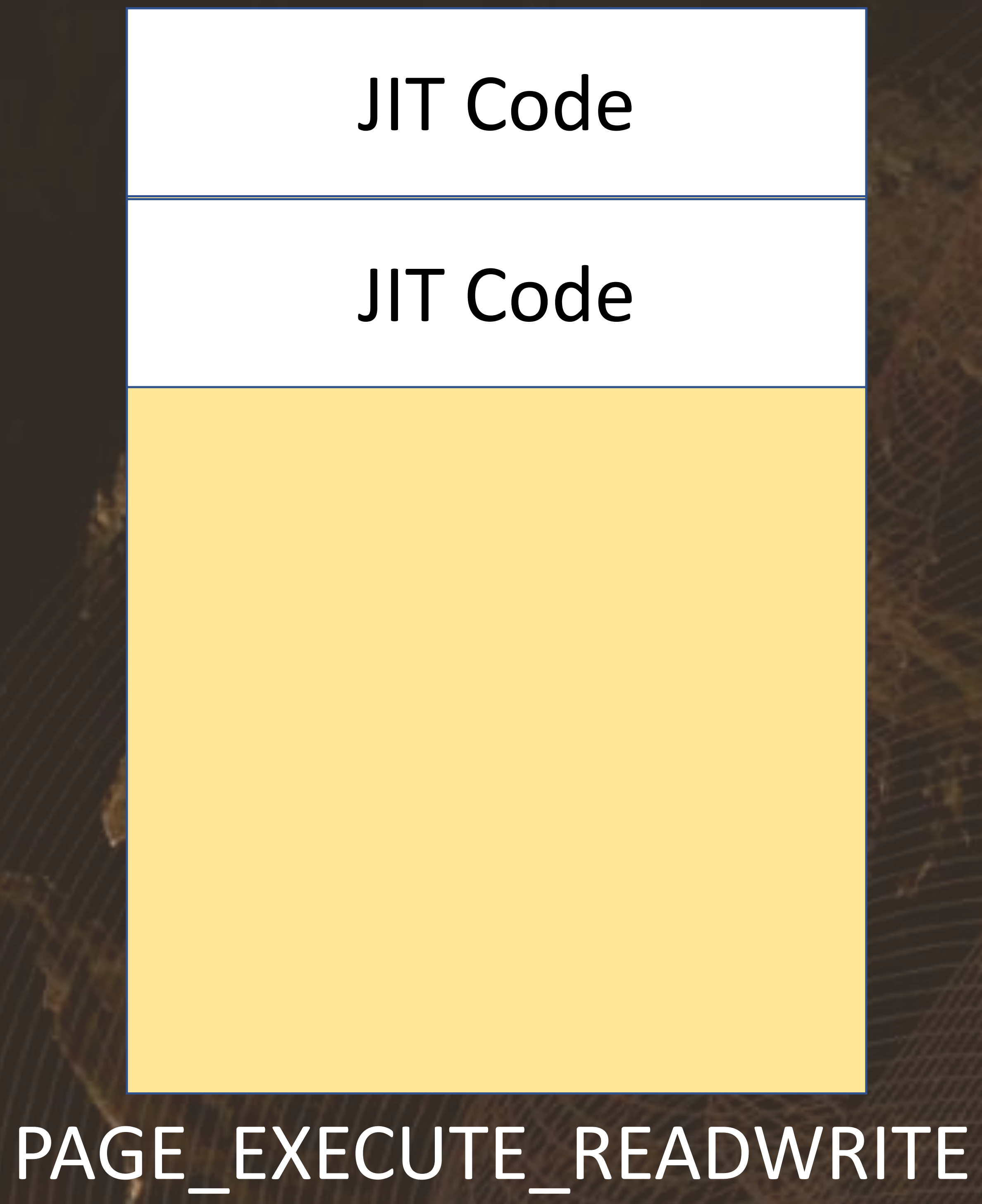


PAGE_EXECUTE_READWRITE

3.1 原始问题

- **Chakra JIT Engine 的内存管理**

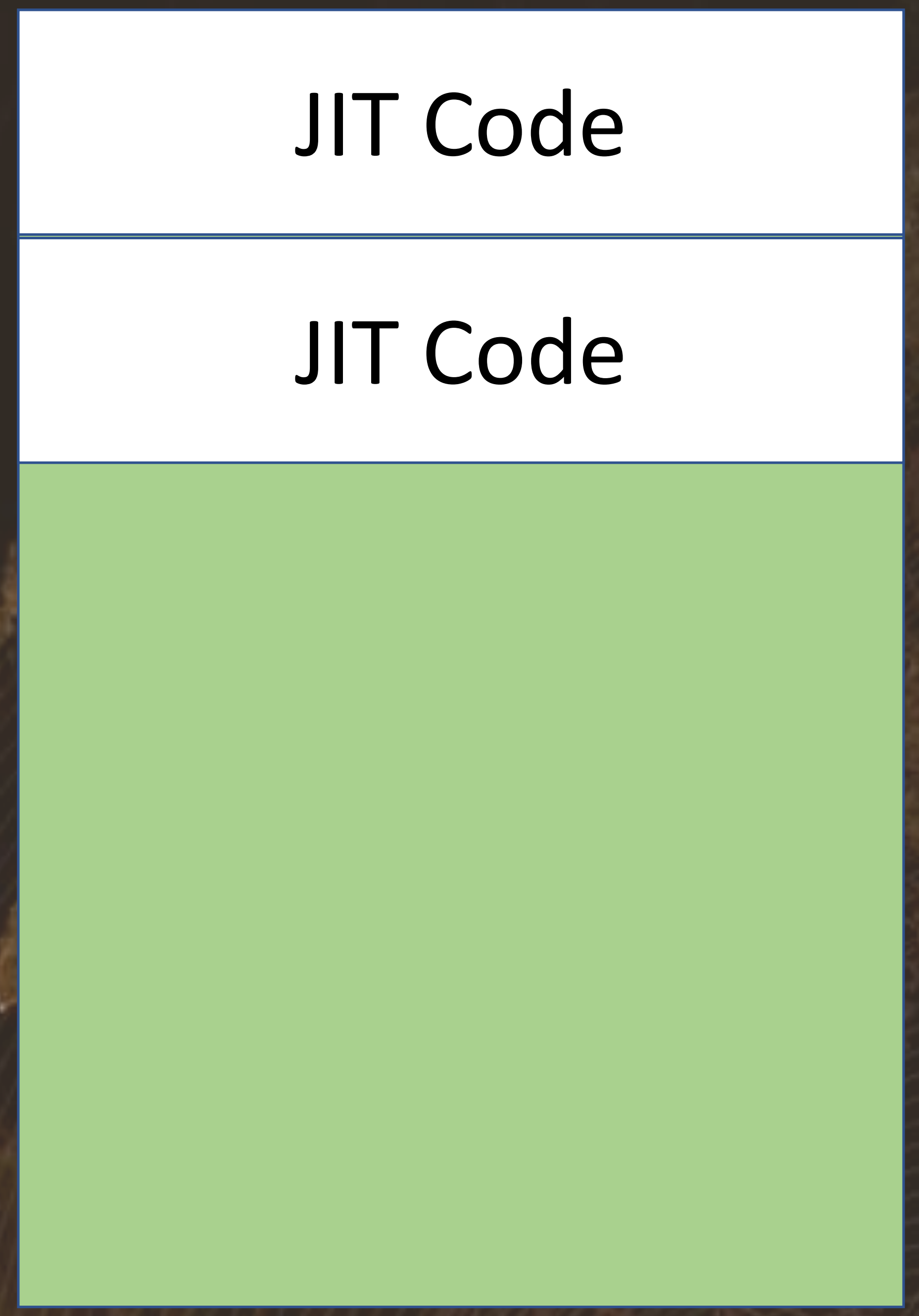
```
Encoder::Encode
  CodeGenWorkItem::RecordNativeCodeSize
    EmitBufferManager::AllocateBuffer
      EmitBufferManager::NewAllocation
        CustomHeap::Heap::Alloc
  CodeGenWorkItem::RecordNativeCode
    EmitBufferManager::CommitBuffer
      CustomHeap::Heap::ProtectAllocationWithExecuteReadWrite
      memcpy_s
      CustomHeap::Heap::ProtectAllocationWithExecuteReadOnly
```



3.1 原始问题

- **Chakra JIT Engine 的内存管理**

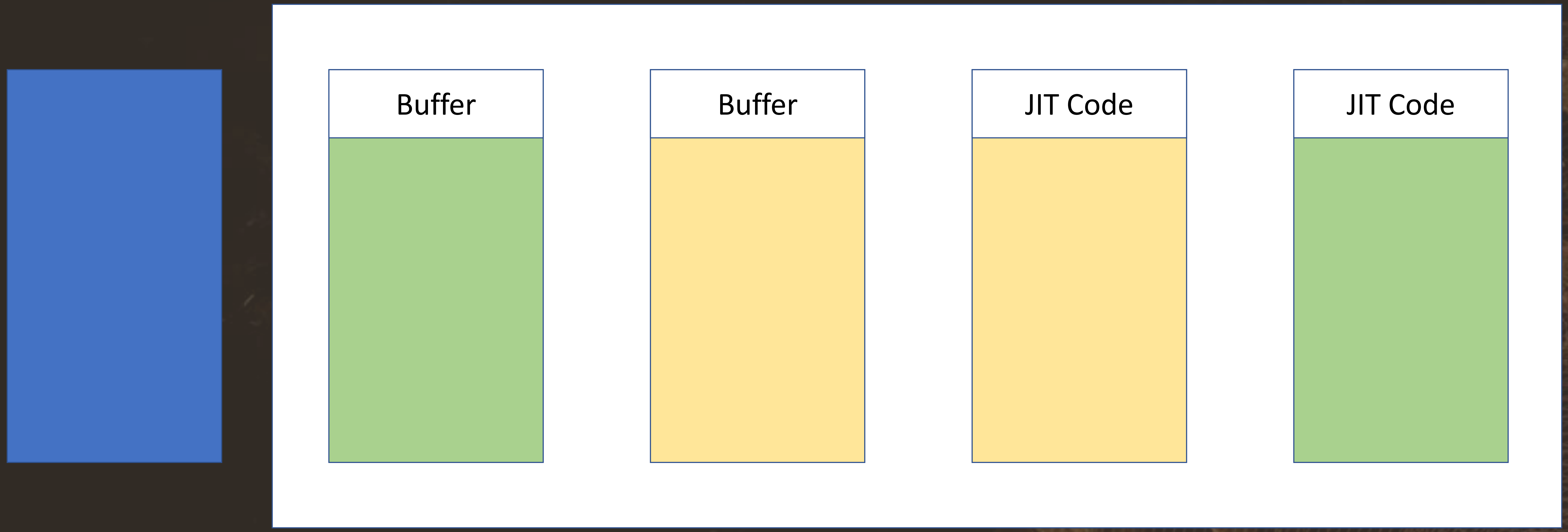
```
Encoder::Encode
  CodeGenWorkItem::RecordNativeCodeSize
    EmitBufferManager::AllocateBuffer
      EmitBufferManager::NewAllocation
        CustomHeap::Heap::Alloc
  CodeGenWorkItem::RecordNativeCode
    EmitBufferManager::CommitBuffer
      CustomHeap::Heap::ProtectAllocationWithExecuteReadWrite
      memcpy_s
      CustomHeap::Heap::ProtectAllocationWithExecuteReadOnly
```



PAGE_EXECUTE

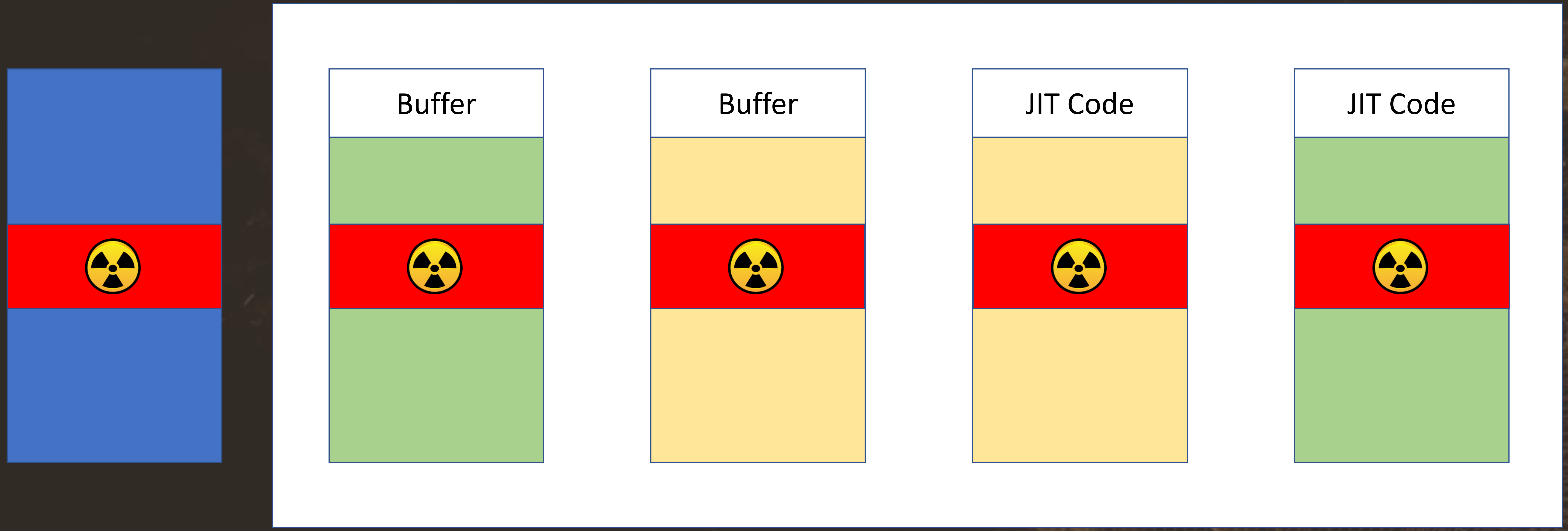
3.1 原始问题

- **Chakra JIT Engine 的内存管理**



3.1 原始问题

- 内存的 Protect 属性是针对整个页面的

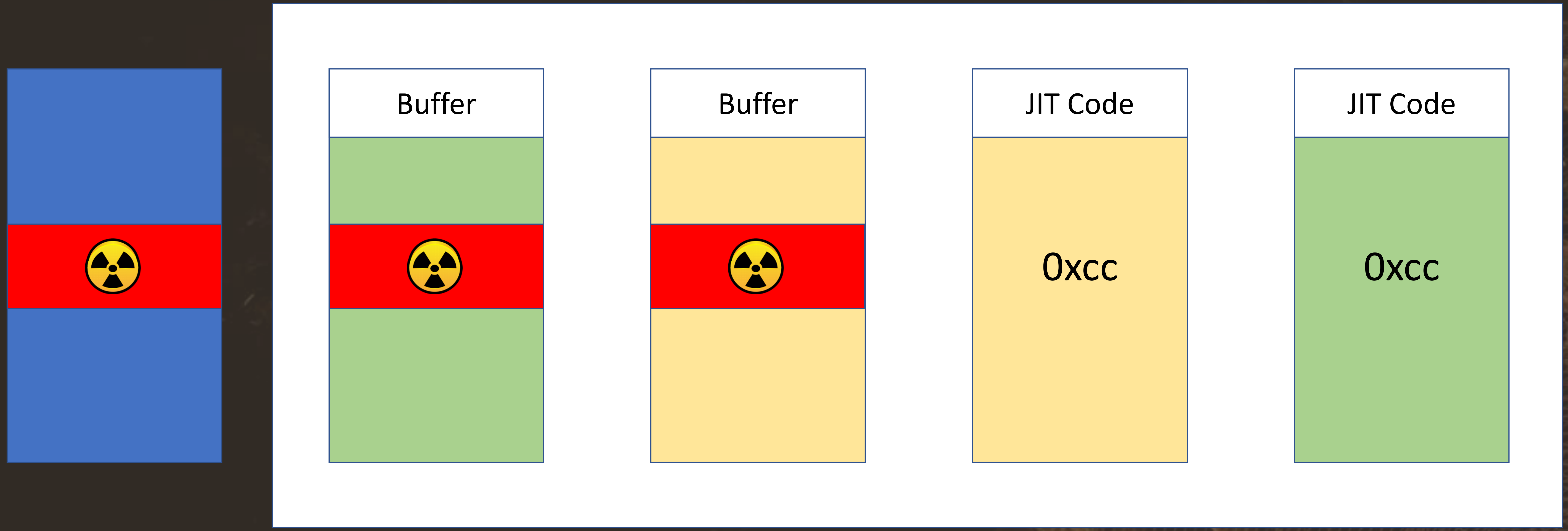


3.1 原始问题

- **利用方案**
  - 读取 JIT 引擎将要使用的下一个页面的地址
  - 将 shellcode 写入该页面的中间部分
  - 触发 JIT 编译
  - 等待 JIT 编译完成
  - 执行 shellcode

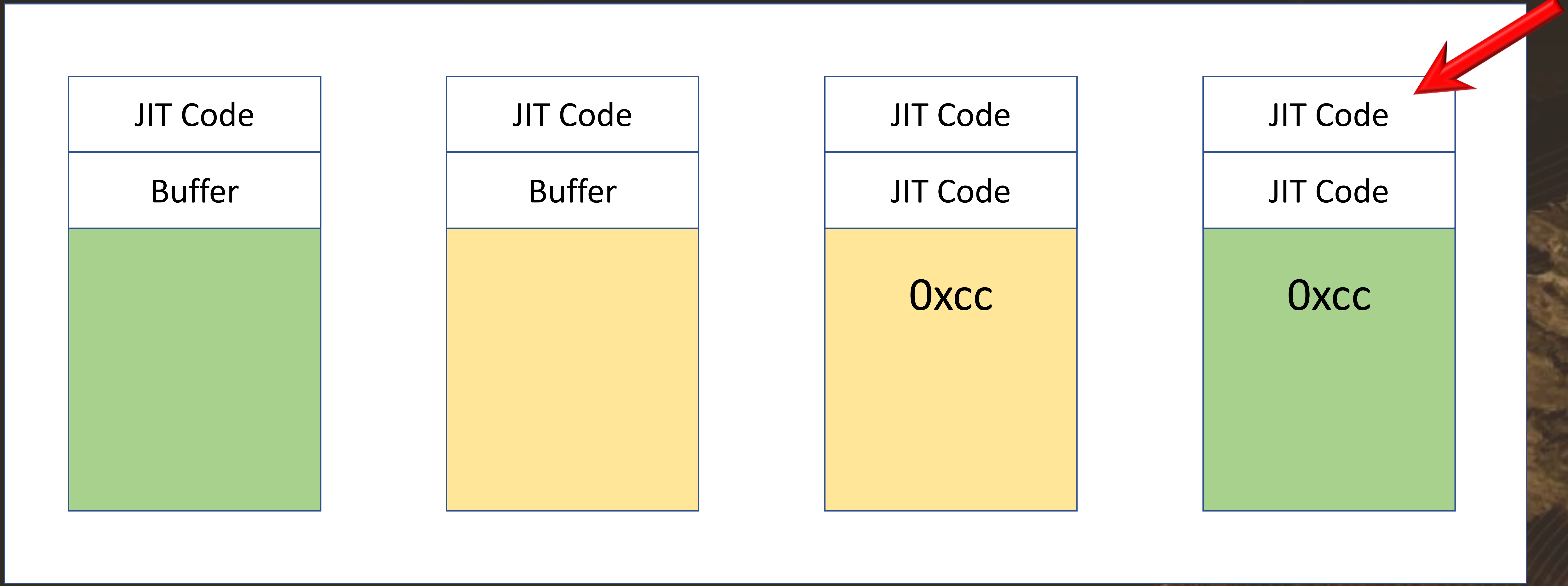
3.2 第一次修复

- Chakra JIT Engine 在拷贝代码后会用 0xcc 填充页面的后续空间



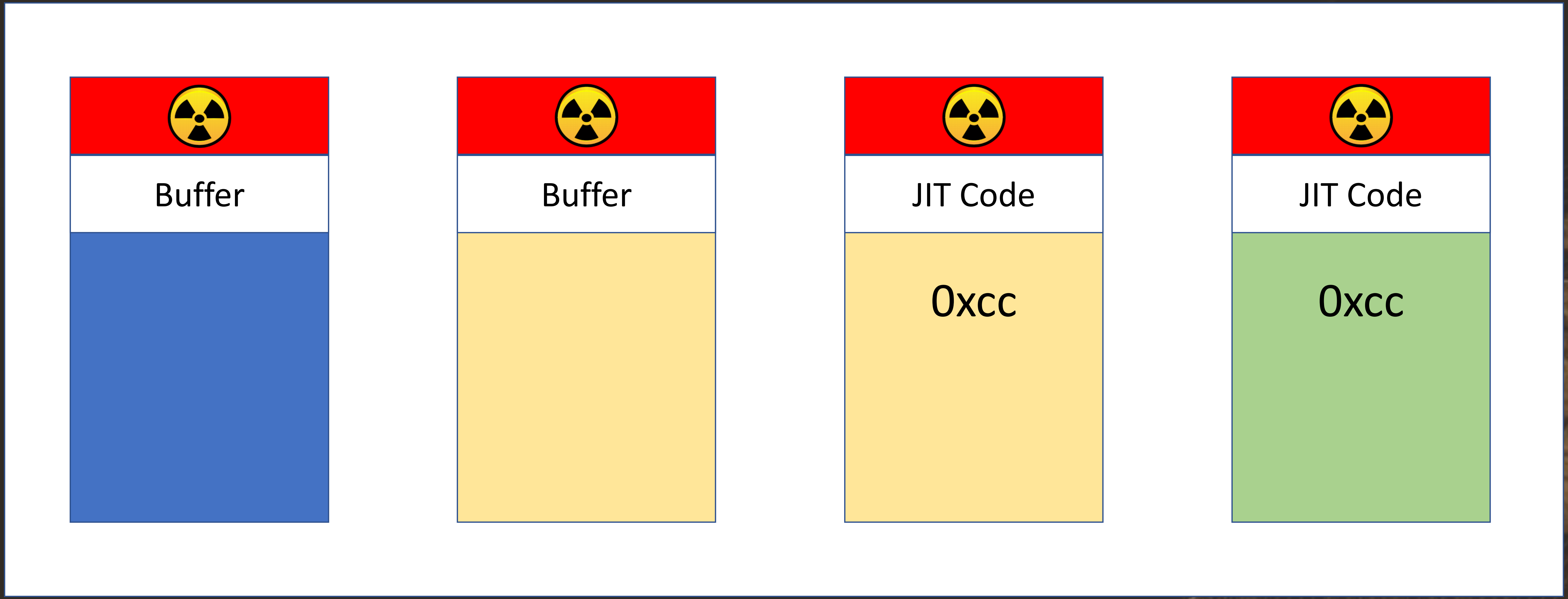
3.3 绕过第一次修复

- Chakra JIT Engine 不能填充页面中已使用的空间



3.3 绕过第一次修复

- 写入页面已使用部分的 shellcode 不会被清除



3.3 绕过第一次修复

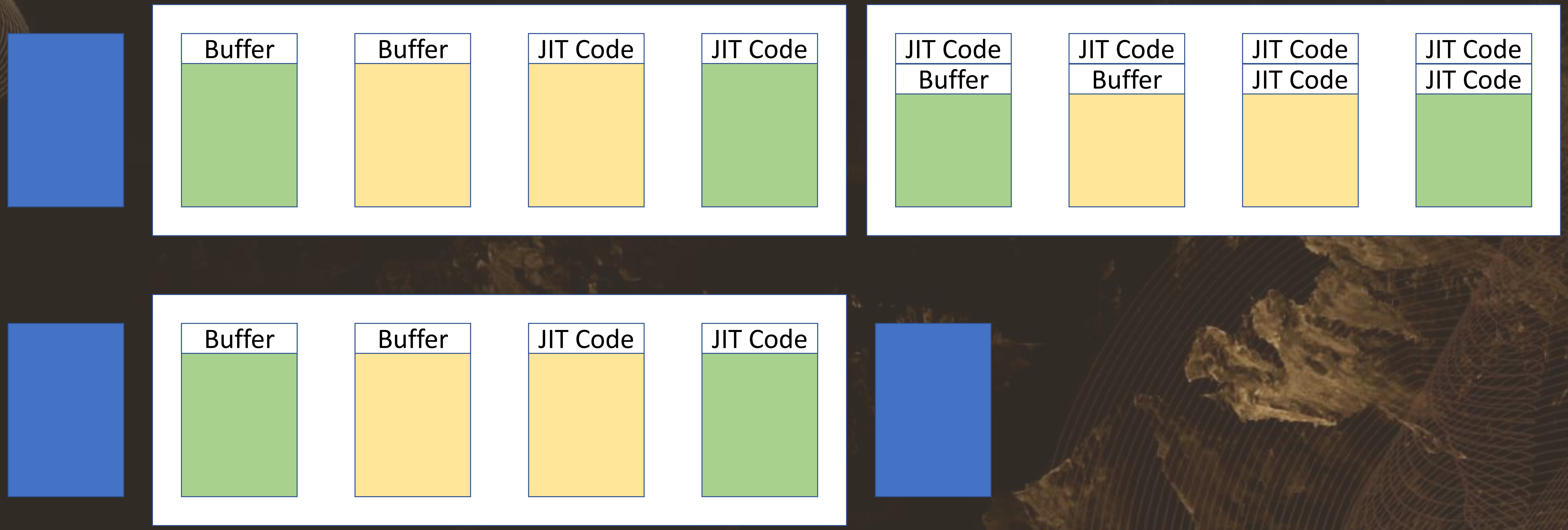
- 利用方案
  - 将一个伪造的页面插入 CustomHeap 的 bucket 中
  - 将 shellcode 写入页面的开始
  - 触发 JIT 编译
  - 等待 JIT 编译完成
  - 执行 shellcode

3.4 第二次修复

- Chakra 在 JIT 引擎中启用了 CFG
  - 分配页面时启用 PAGE_TARGETS_NO_UPDATE
  - 调用 SetProcessValidCallTargets 将 JIT 函数入口设置为有效目标

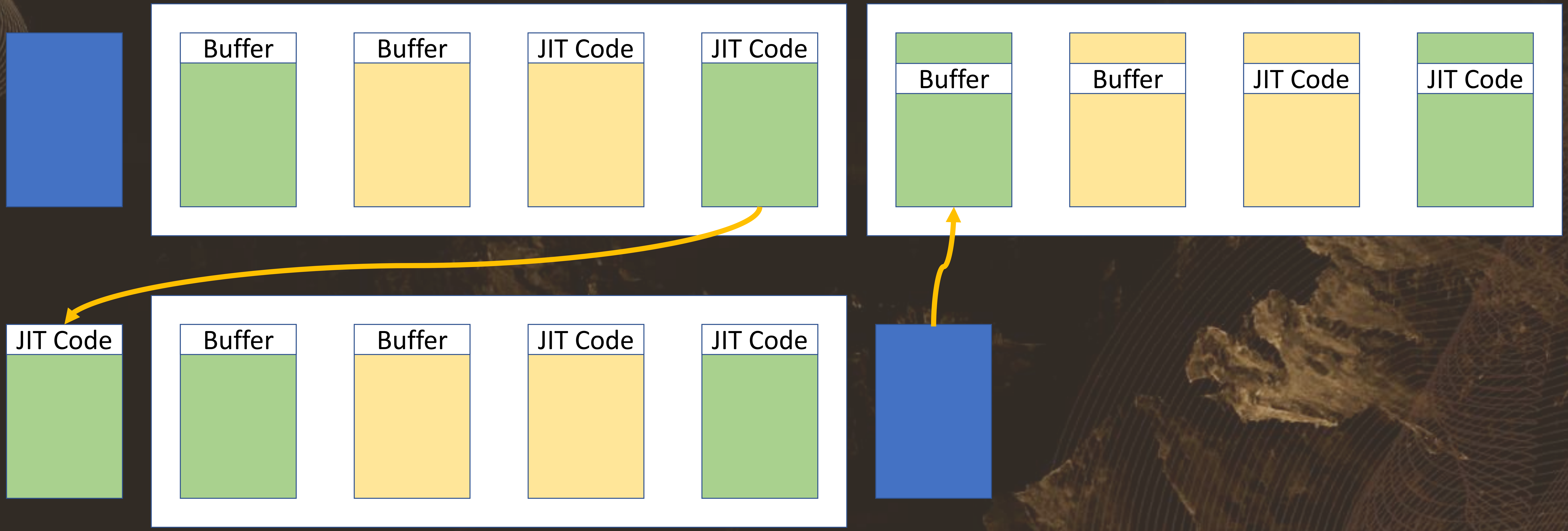
3.5 绕过第二次修复

- 启动第二个 JIT 引擎



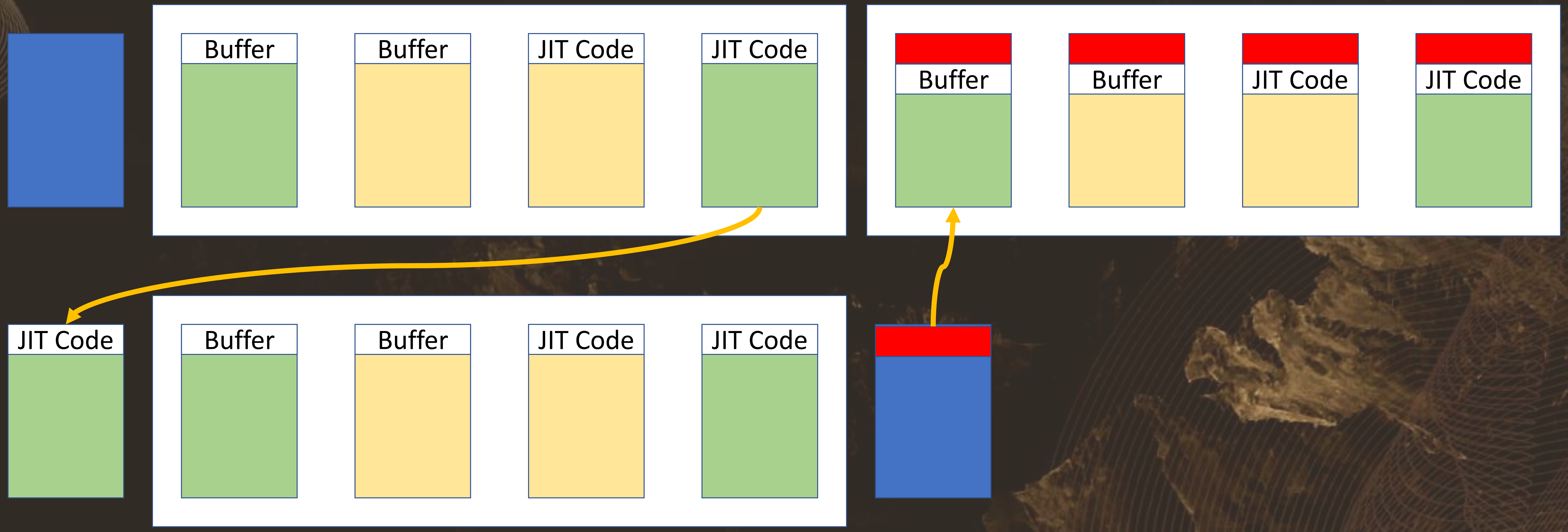
3.5 绕过第二次修复

- 让这两个引擎使用同一个页面



3.5 绕过第二次修复

- **修改生成的 JIT 代码**



3.5 绕过第二次修复

- 利用方案
  - 触发函数 FuncA 的 JIT 编译
  - 读取 FuncA 的 JIT 代码的地址 AddrA
  - 启动第二个 JIT 引擎
  - 修改第二个 JIT 引擎的 CustomHeap 以使用 AddrA
  - 触发第二个 JIT 引擎的 JIT 编译
  - 释放第二个 JIT 引擎
  - 将 shellcode 写入 AddrA
  - 触发第一个 JIT 引擎的 JIT 编译
  - 调用 FuncA 来执行 shellcode

3.6 最终修复

- Out-of-process (OOP) JIT
  - 将整个 JIT 编译的工作放到一个独立的专用进程中
  - 渲染进程不再管理 JIT 引擎使用的内存

Q & A