中国互联网安全领袖峰会
Cyber Security Summit

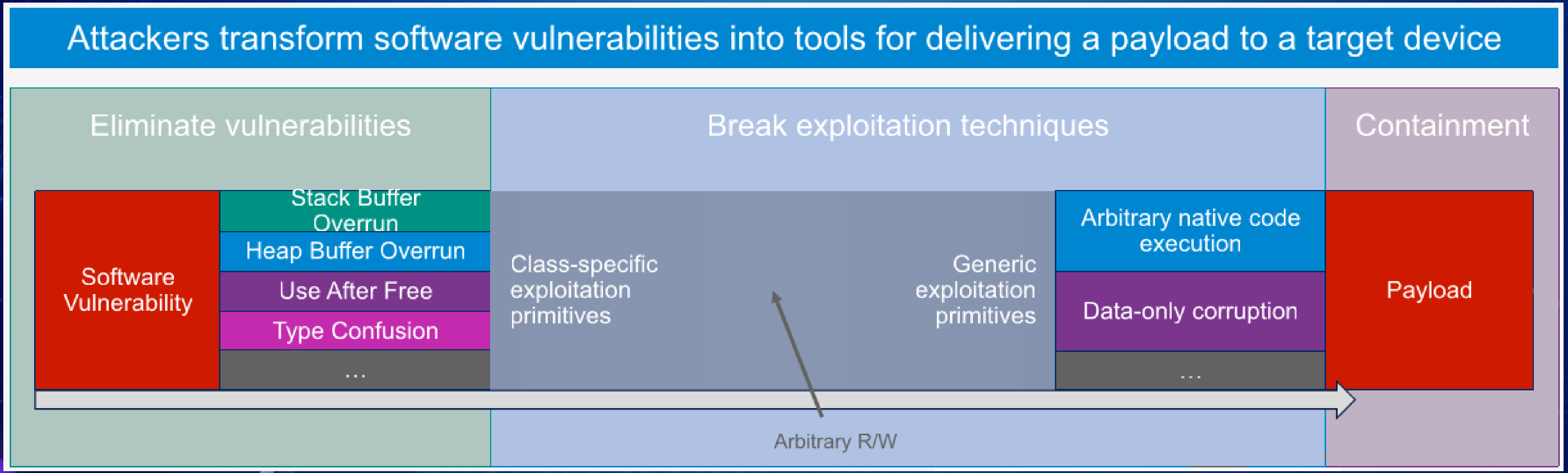
以子之盾，攻子之盾

利用缓解措施自身缺陷突破防线

演讲人：绿盟科技 张云海

Microsoft 防御漏洞利用的策略

缓解措施是其中的一个重要环节



Windows 10 之前的缓解措施

地址空间布局随机化 Address Space Layout Randomization (ASLR)

数据执行保护 Data Execution Prevention (DEP)

堆随机化和元数据保护

SEHOP/SafeSEH

Windows 10 Technical Preview

控制流防护 Control Flow Guard (CFG)

Windows 10 TH1 (1507)

完善控制流防护

- 实现对 JIT 生成代码的防护
- CFG Explicit Suppression

Windows 10 TH2 (1511)

代码完整性防护 Code Integrity Guard （CIG）
映像加载策略

Windows 10 RS1 (1607)

任意代码防护 Arbitrary Code Guard（ACG）

子进程策略

完善控制流防护

- 实现对 longjmp 的防护

Windows 10 RS2 (1703)

完善控制流防护

- Strict Mode CFG
- CFG Export Suppression

引入 CFG Export Suppression 的原因

Compile time

```
void Foo(...) {
   // SomeFunc is address-taken
   // and may be called indirectly
   Object->FuncPtr = SomeFunc;
}
```

Metadata is automatically added to the image which identifies functions that may be called indirectly

```
void Bar(...) {
   // Compiler-inserted check to
   // verify call target is valid
   _guard_check_icall(Object->FuncPtr);
   Object->FuncPtr(xyz);
}
```

A lightweight check is inserted prior to indirect calls which will verify that the call target is valid at runtime

Runtime

Process Start
- Map valid call target data

Image Load
- Update valid call target data with metadata from PE image

Indirect Call
- Perform O(1) validity check
- Terminate process if invalid target
- Jmp if target is valid

CFG is a deterministic mitigation, its security is not dependent on keeping secrets.

For C/C++ code, CFG requires no source code changes.

```
ntdll!LdrpDispatchUserCallTarget:
00007ffb`4e100e10 4c8b1d59e50d00  mov     r11,qword ptr
[ntdll!LdrSystemDllInitBlock+0xb0]
00007ffb`4e100e17 4c8bd0          mov     r10,rax
00007ffb`4e100e1a 49c1ea09        shr     r10,9
00007ffb`4e100e1e 4f8b1cd3        mov     r11,qword ptr [r11+r10*8]
00007ffb`4e100e22 4c8bd0          mov     r10,rax
00007ffb`4e100e25 49c1ea03        shr     r10,3
00007ffb`4e100e29 a80f            test    al,0Fh
00007ffb`4e100e2b 7509            jne     ntdll!LdrpDispatchUserCallTarget+0x26
ntdll!LdrpDispatchUserCallTarget+0x1d:
00007ffb`4e100e2d 4d0fa3d3        bt      r11,r10
00007ffb`4e100e31 7303            jae     ntdll!LdrpDispatchUserCallTarget+0x26
ntdll!LdrpDispatchUserCallTarget+0x23:
00007ffb`4e100e33 48ffe0          jmp     rax
...
```

引入 CFG Export Suppression 的原因

编译时只能确定模块内的间接函数调用

引入 CFG Export Suppression 的原因

编译时只能确定模块内的间接函数调用

跨模块的间接函数调用在运行时才能知晓

引入 CFG Export Suppression 的原因

编译时只能确定模块内的间接函数调用

跨模块的间接函数调用在运行时才能知晓

所有的导出函数都被标记为合法的间接调用目标

引入 CFG Export Suppression 的原因

- 编译时只能确定模块内的间接函数调用
- 跨模块的间接函数调用在运行时才能知晓
- 所有的导出函数都被标记为合法的间接调用目标
- 通过 Explicit Suppression 来排除敏感函数有局限性

CFG Export Suppression 如何工作

编译时：设置 GuardFlags 标志位

IMAGE_GUARD_CF_INSTRUMENTED	0x00000100
IMAGE_GUARD_CFW_INSTRUMENTED	0x00000200
IMAGE_GUARD_CF_FUNCTION_TABLE_PRESENT	0x00000400
IMAGE_GUARD_SECURITY_COOKIE_UNUSED	0x00000800
IMAGE_GUARD_PROTECT_DELAYLOAD_IAT	0x00001000
IMAGE_GUARD_DELAYLOAD_IAT_IN_ITS_OWN_SECTION	0x00002000
IMAGE_GUARD_CF_EXPORT_SUPPRESSION_INFO_PRESENT	0x00004000
IMAGE_GUARD_CF_ENABLE_EXPORT_SUPPRESSION	0x00008000
IMAGE_GUARD_CF_LONGJUMP_TABLE_PRESENT	0x00010000

CFG Export Suppression 如何工作

编译时：生成 GuardCFFunctionTable

```
u_long ntohl(
  u_long netlong
) {

...

}
```

```
dd rva WSCInstallProviderAndChains64_32
db 0
dd rva WSCSetApplicationCategory
db 0
dd rva ntohl
db 0
dd rva ntohs
db 0
dd rva PathAddBackslashA
db 2
dd rva SHRegQueryUSValueA
db 2
dd rva RaiseFailFastException
db 2
dd rva CheckIfStateChangeNotificationExists
db 2
dd rva GetCalendarInfoW
```

CFG Export Suppression 如何工作

编译时：生成 GuardCFFunctionTable

```
BOOL WINAPI SetThreadContext(
  _In_       HANDLE  hThread,
  _In_ const CONTEXT *lpContext
) {

...

}
```

```
dd rva LoadStringA
db 2
dd rva CreatePrivateObjectSecurity
db 2
dd rva SetThreadContext
db 1
dd rva PackageFullNameFromId
db 2
dd rva GetSystemWow64DirectoryW
db 2
dd rva GetSystemWow64Directory2W
db 2
dd rva GetWindowsDirectoryW
db 2
dd rva GetSystemWindowsDirectoryW
db 2
dd rva Wow64SetThreadDefaultGuestMachine
```

CFG Export Suppression 如何工作

编译时：生成 GuardCFFunctionTable

```
BOOL WINAPI IsWow64Process(
  _In_       HANDLE hProcess,
  _Out_      PBOOL Wow64Process
) {

...

}
```

```
dd rva VirtualFreeEx
db 2
dd rva WerGetFlags
db 2
dd rva IsWow64Process
db 2
dd rva ReadProcessMemory
db 2
dd rva GetFileType
db 2
dd rva EnumSystemGeoID
db 2
dd rva LocaleNameToLCID
db 2
dd rva EnumDateFormatsExEx
db 2
dd rva Internal_EnumDateFormats
db 2
```

CFG Export Suppression 如何工作

编译时：在间接函数调用前插入检查

```
BOOL isWow64 = FALSE;

fnIsWow64Process = GetProcAddress(
    GetModuleHandleW(L"kernel32"), "IsWow64Process");

fnIsWow64Process(GetCurrentProcess(), &isWow64);
```

```
mov     [rsp+48h+var_28], 0
lea     rcx, ModuleName ; "kernel32"
call    cs:__imp_GetModuleHandleW
lea     rdx, ProcName    ; "IsWow64Process"
mov     rcx, rax         ; hModule
call    cs:__imp_GetProcAddress
mov     [rsp+48h+var_18], rax
call    cs:__imp_GetCurrentProcess
mov     [rsp+48h+var_10], rax
mov     rcx, [rsp+48h+var_18]
mov     [rsp+48h+Target], rcx
mov     rcx, [rsp+48h+Target] ; Target
call    cs:__guard_check_icall_fptr
lea     rdx, [rsp+48h+var_28]
mov     rax, [rsp+48h+var_10]
mov     rcx, rax
call    [rsp+48h+Target]
```

CFG Export Suppression 如何工作

加载时：根据 GuardFlags 更新函数指针__guard_check_icall_fptr

```
if ( !LdrControlFlowGuardEnforcedWithExportSuppression()
  || (fptr = LdrpValidateUserCallTargetES, !(*(LoadConfig + 0x90) & 0x4000)) )
{
  fptr = LdrpValidateUserCallTarget;
}
*__guard_check_icall_fptr = fptr;
```

CFG Export Suppression 如何工作

加载时：根据 GuardFlags 更新函数指针__guard_dispatch_icall_fptr

```
if ( !LdrControlFlowGuardEnforcedWithExportSuppression()
  || (fptr = LdrpDispatchUserCallTargetES, !(*(LoadConfig + 0x90) & 0x4000)) )
{
  fptr = LdrpDispatchUserCallTarget;
}
*_guard_dispatch_icall_fptr = fptr;
```

CFG Export Suppression 如何工作

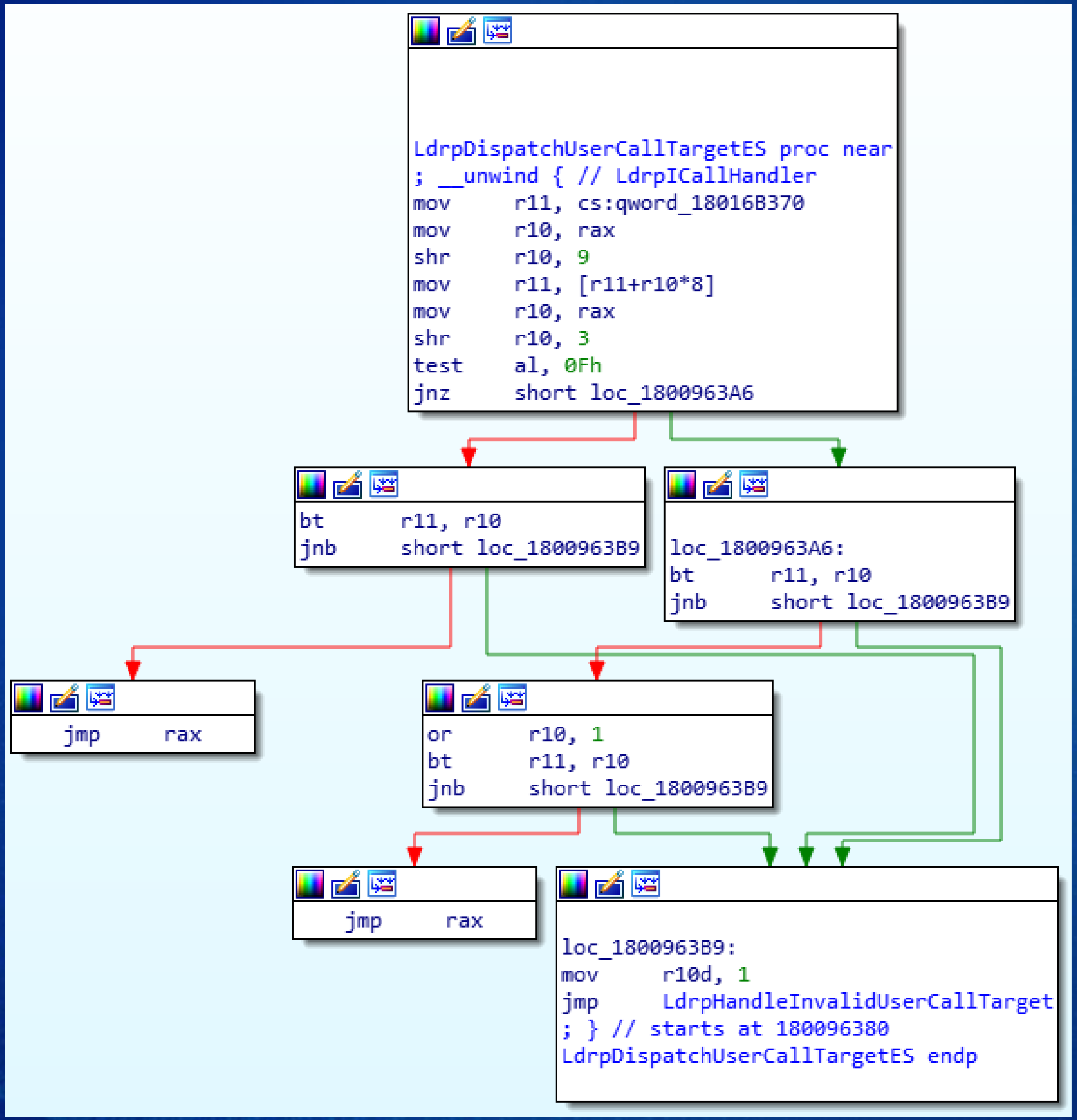
加载时：根据 GuardCFFunctionTable 更新 CFG Bitmap

ntohl	7fff344afa90
SetThreadContext	7fff34437aa0
IsWow64Process	7fff34488590

7ff5fc700de8	0001000000000000000000000010000000011000000000000000100000000000001
7ff5fc702210	0000000000001000000000000000000000000000000000000000000000000000
7ff5fc702be8	0000000000000000000000000000000000000010000101010101010101010101

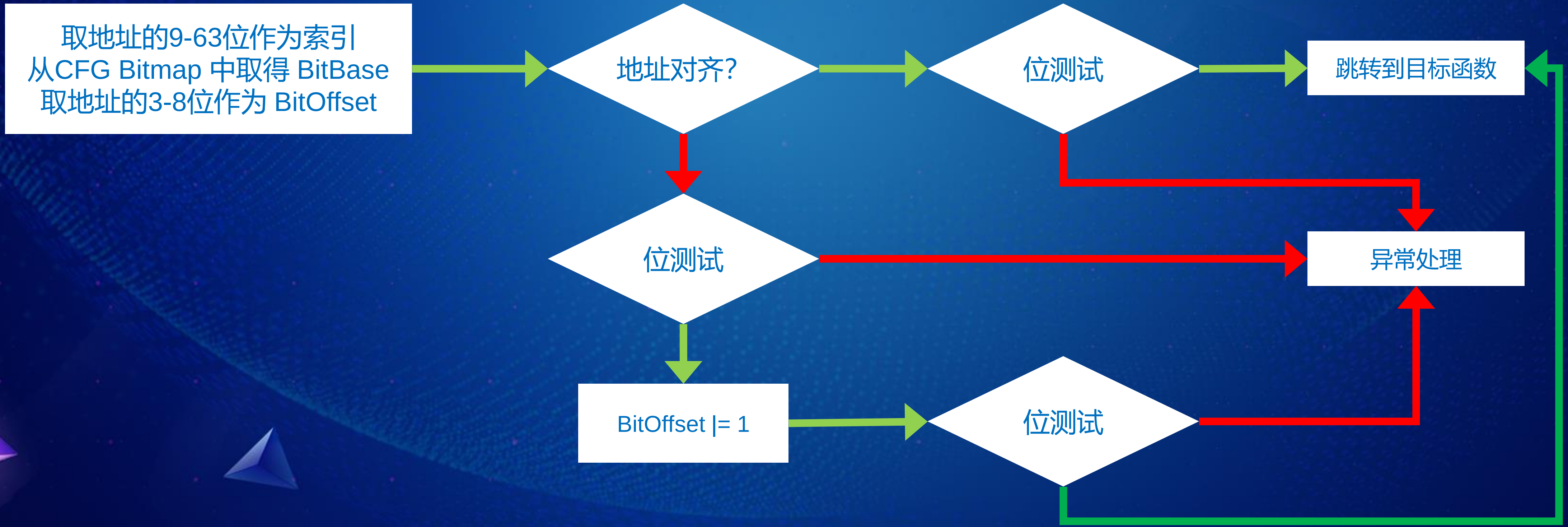
CFG Export Suppression 如何工作

运行时：LdrpDispatchUserCallTargetES 的检查逻辑


LdrpDispatchUserCallTargetES proc near
; __unwind { // LdrpICallHandler
mov r11, cs:qword_18016B370
mov r10, rax
shr r10, 9
mov r11, [r11+r10*8]
mov r10, rax
shr r10, 3
test al, 0Fh
jnz short loc_1800963A6
bt r11, r10
jnb short loc_1800963B9
loc_1800963A6:
bt r11, r10
jnb short loc_1800963B9
jmp rax
or r10, 1
bt r11, r10
jnb short loc_1800963B9
jmp rax
loc_1800963B9:
mov r10d, 1
jmp LdrpHandleInvalidUserCallTarget
; } // starts at 180096380
LdrpDispatchUserCallTargetES endp

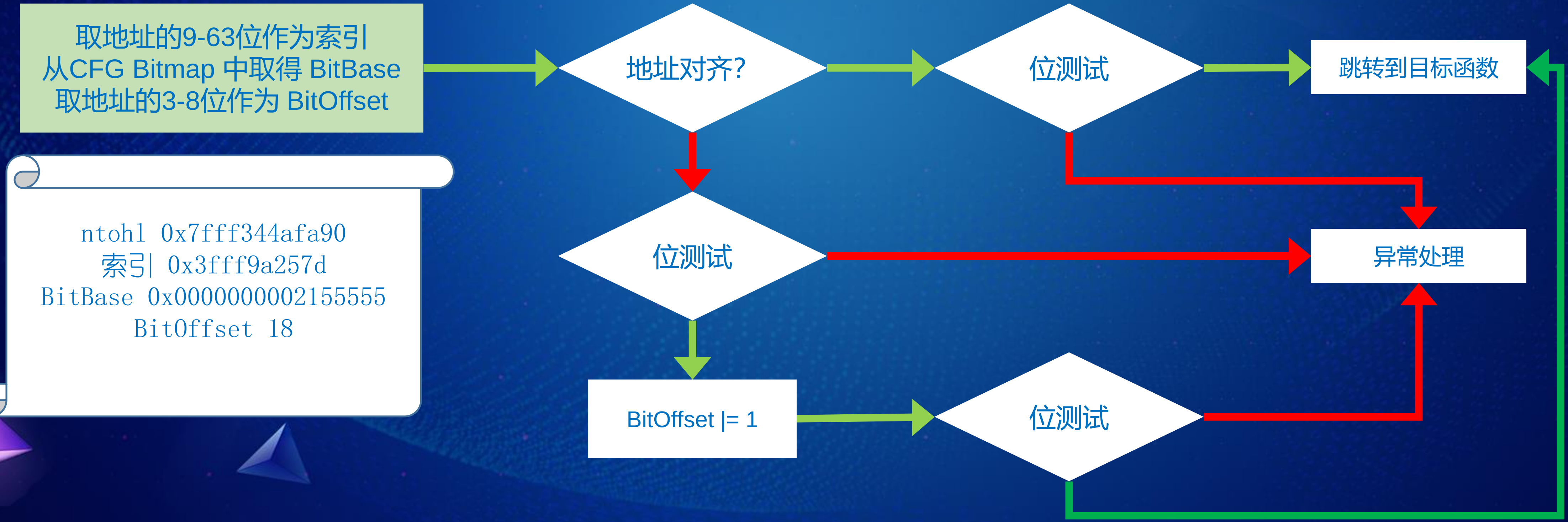
CFG Export Suppression 如何工作

运行时：LdrpDispatchUserCallTargetES 的检查逻辑



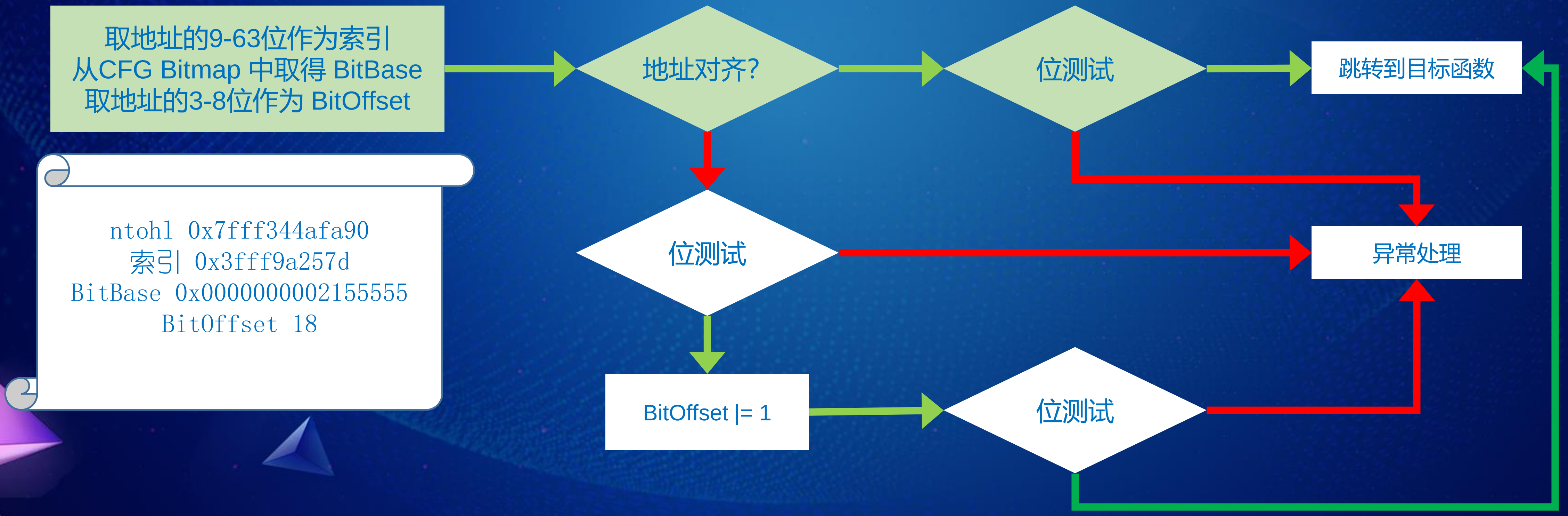
CFG Export Suppression 如何工作

运行时：LdrpDispatchUserCallTargetES 的检查逻辑



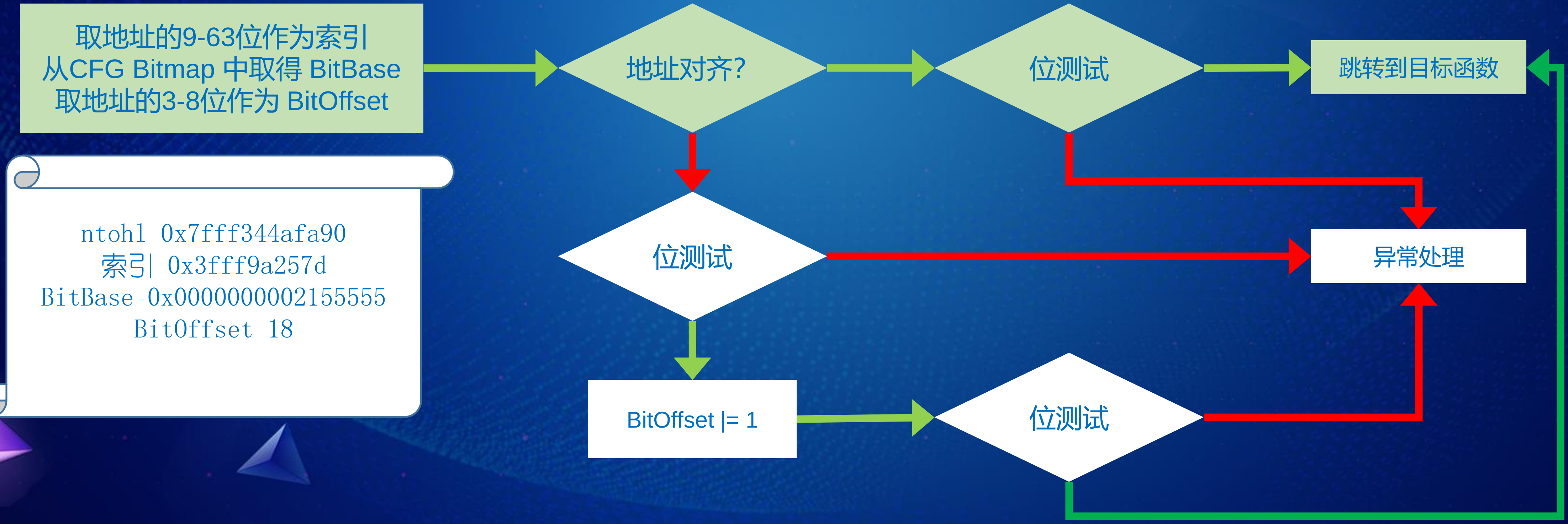
CFG Export Suppression 如何工作

运行时：LdrpDispatchUserCallTargetES 的检查逻辑



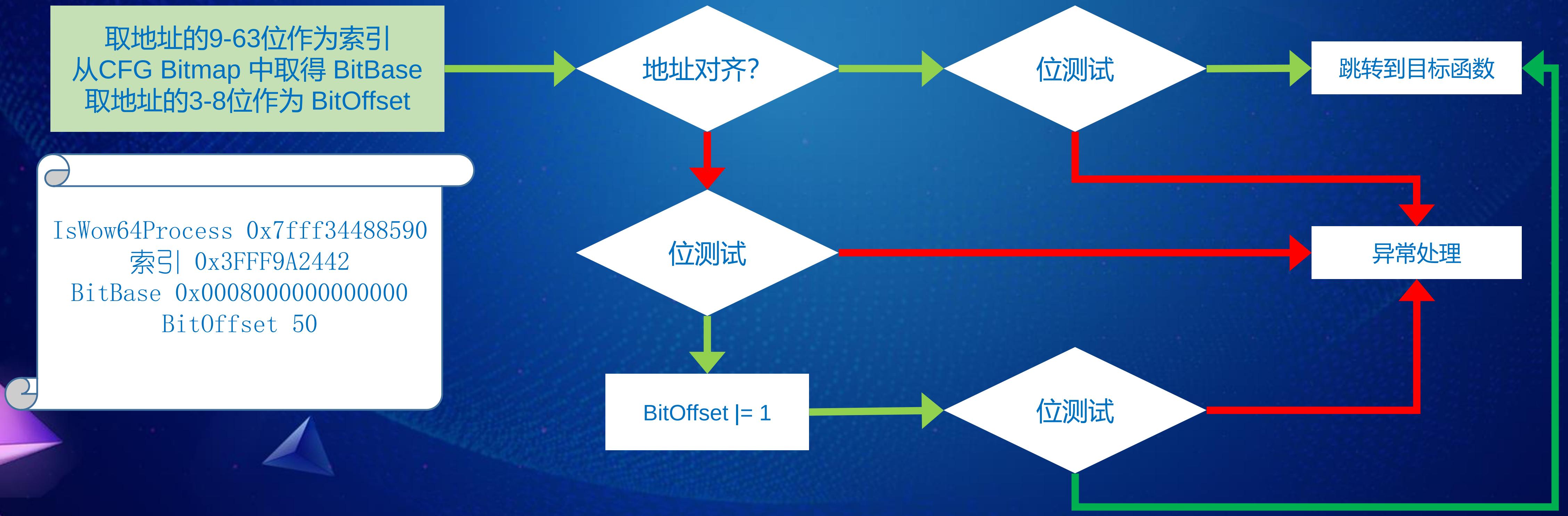
CFG Export Suppression 如何工作

运行时：LdrpDispatchUserCallTargetES 的检查逻辑



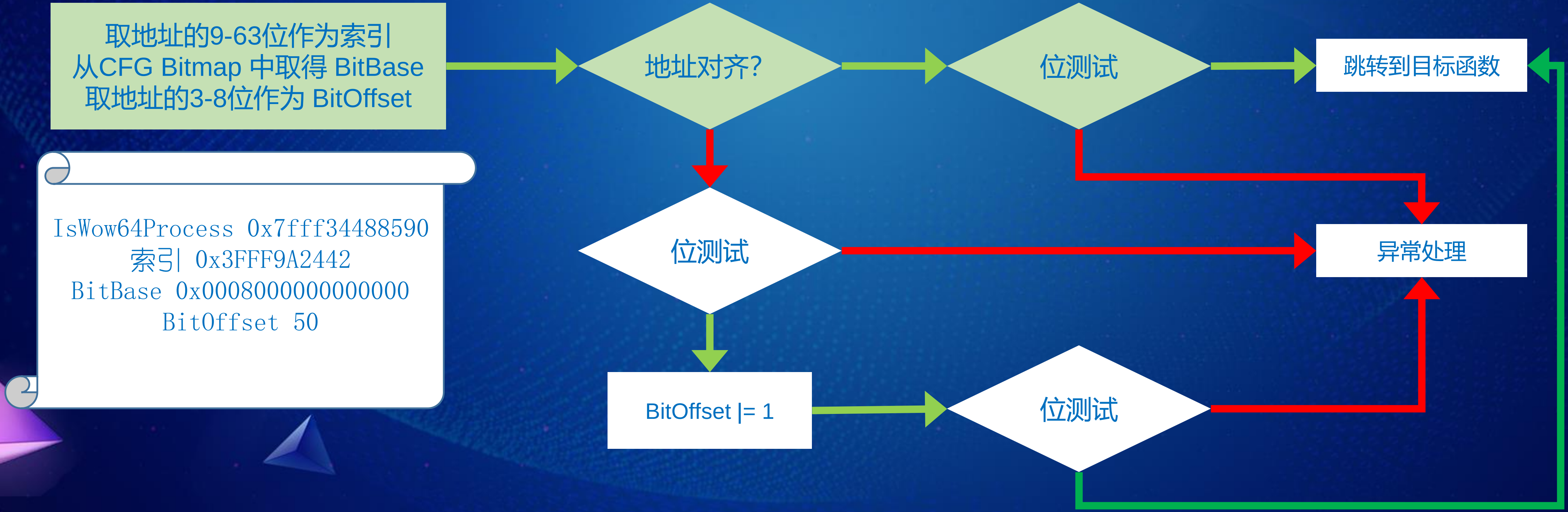
CFG Export Suppression 如何工作

运行时：LdrpDispatchUserCallTargetES 的检查逻辑



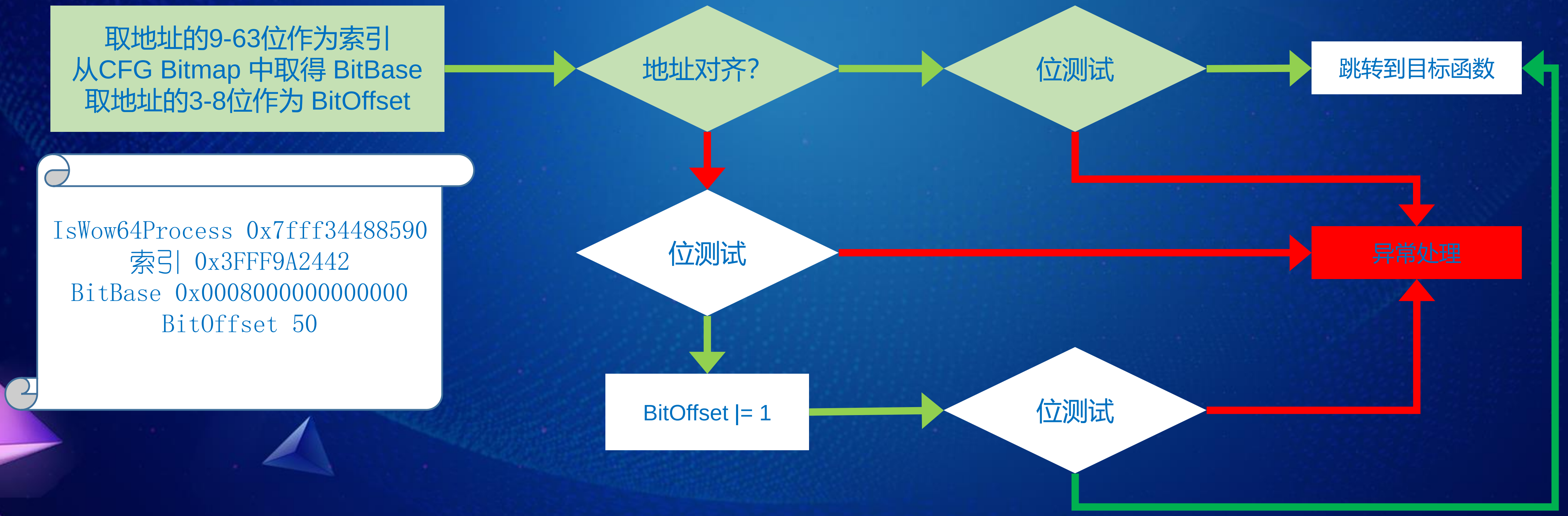
CFG Export Suppression 如何工作

运行时：LdrpDispatchUserCallTargetES 的检查逻辑



CFG Export Suppression 如何工作

运行时：LdrpDispatchUserCallTargetES 的检查逻辑



CFG Export Suppression 如何工作

运行时：在 GetProcAddress 函数中更新 CFG Bitmap

- KERNELBASE!GetProcAddress
- ntdll!LdrGetProcedureAddressForCaller
- ntdll!RtlGuardGrantSuppressedCallAccess
- ntdll!RtlpGuardGrantSuppressedCallAccess
- ntdll!NtSetInformationVirtualMemory

CFG Export Suppression 如何工作

运行时：在 GetProcAddress 函数中更新 CFG Bitmap

```
7fff344afa90 ntohl
7fff34437aa0 SetThreadContext
7fff34488590 IsWow64Process
```

```
7ff5fc700de8   0001000000000000000000000010000000001100000000000001000000000001

7ff5fc702210   0000000000001000000000000000000000000000000000000000000000000000

7ff5fc702be8   0000000000000000000000000000000000000000100001010101010101010101
```

CFG Export Suppression 如何工作

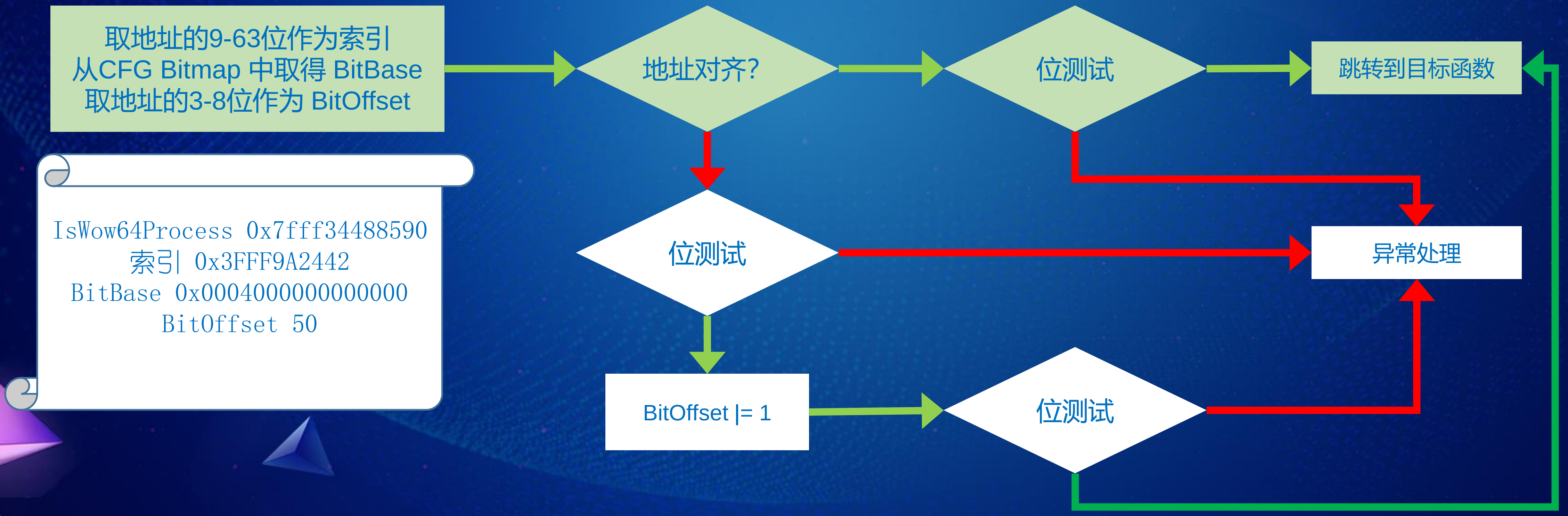
运行时：在 GetProcAddress 函数中更新 CFG Bitmap

```
7fff344afa90 ntohl
7fff34437aa0 SetThreadContext
7fff34488590 IsWow64Process
```

```
7ff5fc700de8   0001000000000000000000000010000000011000000000000010000000000001

7ff5fc702210   0000000000000100000000000000000000000000000000000000000000000000

7ff5fc702be8   0000000000000000000000000000000000000010000101010101010101010101
```

CFG Export Suppression 如何工作

运行时：调用 GetProcAddress 之后的调用



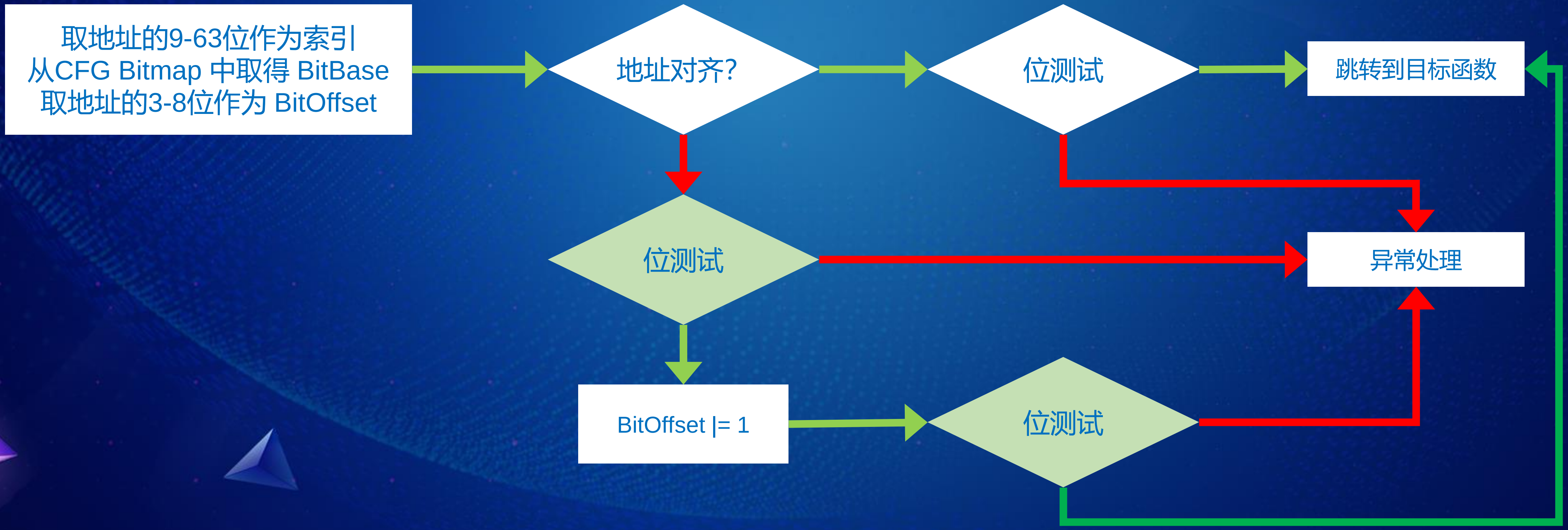
CFG Export Suppression 的问题

函数地址对齐时，每个函数需要占用 CFG Bitmap 中的两位

- 偶数位表示该函数可以被间接调用
- 奇数位表示该函数是被抑制的导出函数

CFG Export Suppression 的问题

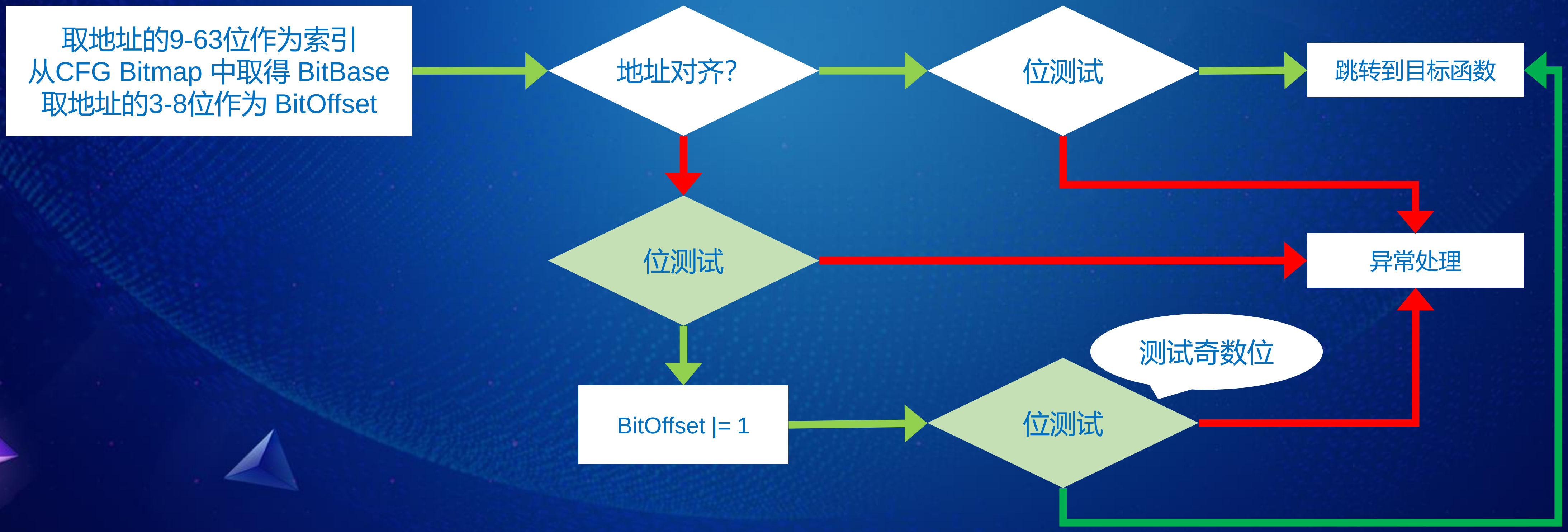
函数地址未对齐时，使用 CFG Bitmap 中同样的两位



中国互联网安全领袖峰会
Cyber Security Summit

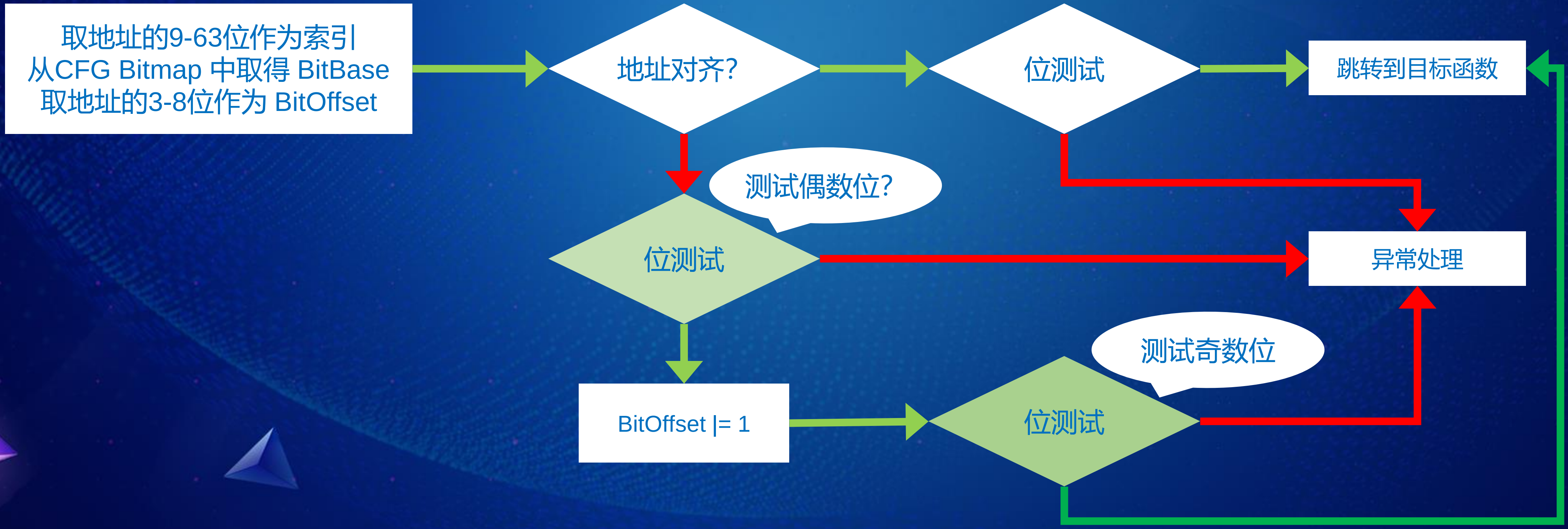
CFG Export Suppression 的问题

函数地址未对齐时，使用 CFG Bitmap 中同样的两位



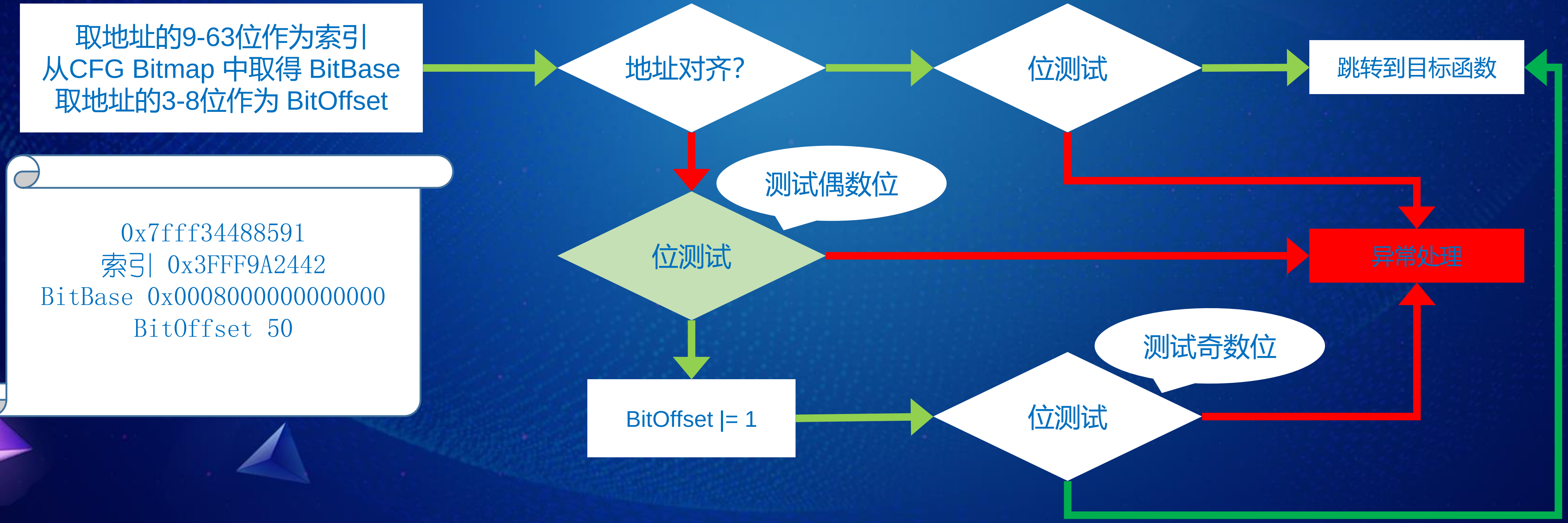
CFG Export Suppression 的问题

函数地址未对齐时，使用 CFG Bitmap 中同样的两位



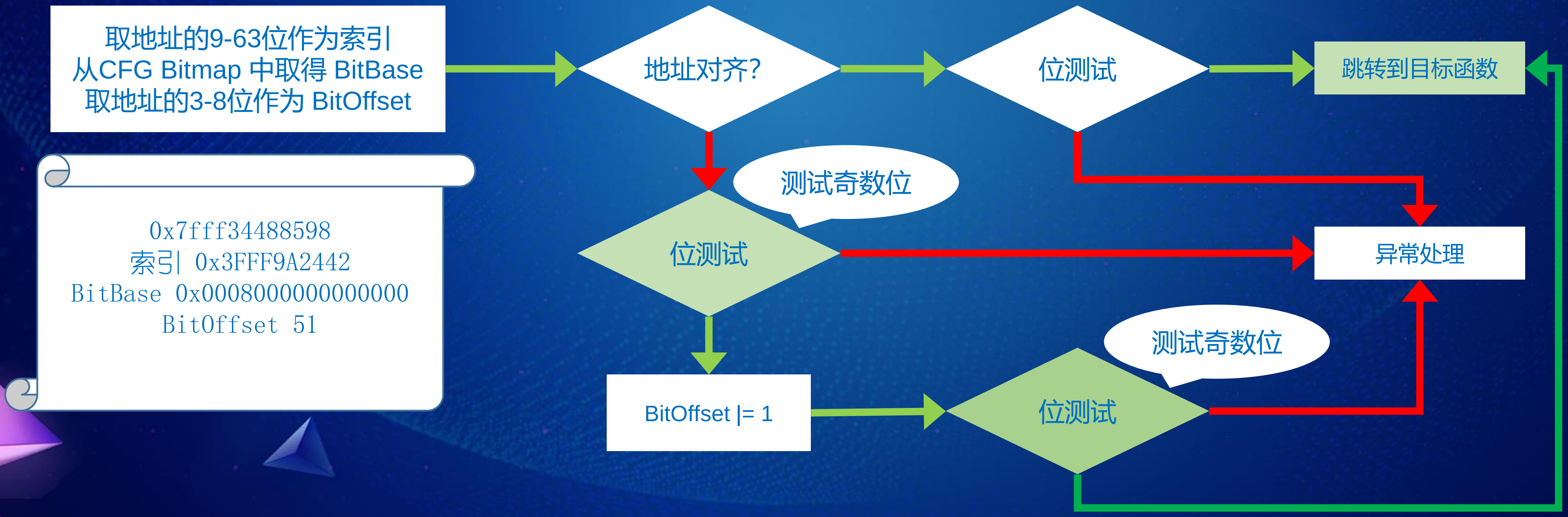
CFG Export Suppression 的问题

函数地址未对齐时，使用 CFG Bitmap 中同样的两位



CFG Export Suppression 的问题

函数地址未对齐时，使用 CFG Bitmap 中同样的两位



CFG Export Suppression 的问题

- ✔ 导出函数本身不可以被间接调用

CFG Export Suppression 的问题

- ✔ 导出函数本身不可以被间接调用
- ✘ 后续的15个地址中有8个可以被间接调用

利用技巧

特殊的导出函数 — 系统调用

```
ntdll!NtContinue:
00007ff9`ef170690 4c8bd1          mov     r10,rcx
00007ff9`ef170693 b843000000      mov     eax,43h
00007ff9`ef170698 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff9`ef1706a0 7503            jne     ntdll!NtContinue+0x15 (00007ff9`ef1706a5)
00007ff9`ef1706a2 0f05            syscall
00007ff9`ef1706a4 c3              ret
00007ff9`ef1706a5 cd2e            int     2Eh
00007ff9`ef1706a7 c3              ret
00007ff9`ef1706a8 0f1f840000000000 nop     dword ptr [rax+rax]
ntdll!NtQueryDefaultUILanguage:
00007ff9`ef1706b0 4c8bd1          mov     r10,rcx
00007ff9`ef1706b3 b844000000      mov     eax,44h
00007ff9`ef1706b8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff9`ef1706c0 7503            jne     ntdll!NtQueryDefaultUILanguage+0x15 (00007ff9`ef1706c5)
00007ff9`ef1706c2 0f05            syscall
00007ff9`ef1706c4 c3              ret
00007ff9`ef1706c5 cd2e            int     2Eh
00007ff9`ef1706c7 c3              ret
00007ff9`ef1706c8 0f1f840000000000 nop     dword ptr [rax+rax]
```

利用技巧

选择恰当的偏移可以“滑动”到下一函数

```
ntdll!NtConvertBetweenAuxiliaryCounterAndPerformanceCounter+0xe:
77ee330e d40f                 aamb    0Fh
77ee3310 34c3                 xor     al,0C3h
77ee3312 8da42400000000       lea     esp,[esp]
77ee3319 8da42400000000       lea     esp,[esp]
ntdll!NtContinue:
77ee3320 b87e010000           mov     eax,17Eh
77ee3325 e803000000           call    ntdll!NtContinue+0xd (77ee332d)
77ee332a c20800               ret     8
77ee332d 8bd4                 mov     edx,esp
77ee332f 0f34                 sysenter
77ee3331 c3                   ret
77ee3332 8da42400000000       lea     esp,[esp]
77ee3339 8da42400000000       lea     esp,[esp]
```

利用技巧

选择恰当的偏移可以“滑动”到下一函数

```
ntdll!NtDuplicateToken+0xe:
00007ffd`2d20a7ce 7f01           jg      ntdll!NtDuplicateToken+0x11 (00007ffd`2d20a7d1)
00007ffd`2d20a7d0 7503           jne     ntdll!NtDuplicateToken+0x15 (00007ffd`2d20a7d5)
00007ffd`2d20a7d2 0f05           syscall
00007ffd`2d20a7d4 c3             ret
00007ffd`2d20a7d5 cd2e           int     2Eh
00007ffd`2d20a7d7 c3             ret
```

利用技巧

选择恰当的偏移可以“滑动”到下一函数

```
ntdll!NtDuplicateToken+0xe:
00007ffd`2d20a7ce 7f01            jg      ntdll!NtDuplicateToken+0x11 (00007ffd`2d20a7d1)
ntdll!NtDuplicateToken+0x11:
00007ffd`2d20a7d1 030f            add     ecx,dword ptr [rdi]
00007ffd`2d20a7d3 05c3cd2ec3      add     eax,0C32ECDC3h
00007ffd`2d20a7d8 0f1f840000000000 nop     dword ptr [rax+rax]
ntdll!NtContinue:
00007ffd`2d20a7e0 4c8bd1          mov     r10,rcx
00007ffd`2d20a7e3 b843000000      mov     eax,43h
00007ffd`2d20a7e8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffd`2d20a7f0 7503            jne     ntdll!NtContinue+0x15 (00007ffd`2d20a7f5)
00007ffd`2d20a7f2 0f05            syscall
00007ffd`2d20a7f4 c3              ret
00007ffd`2d20a7f5 cd2e            int     2Eh
00007ffd`2d20a7f7 c3              ret
```

利用技巧

利用这一技巧可以间接调用大部分的系统调用
包括危险的敏感函数，如 NtContinue 等

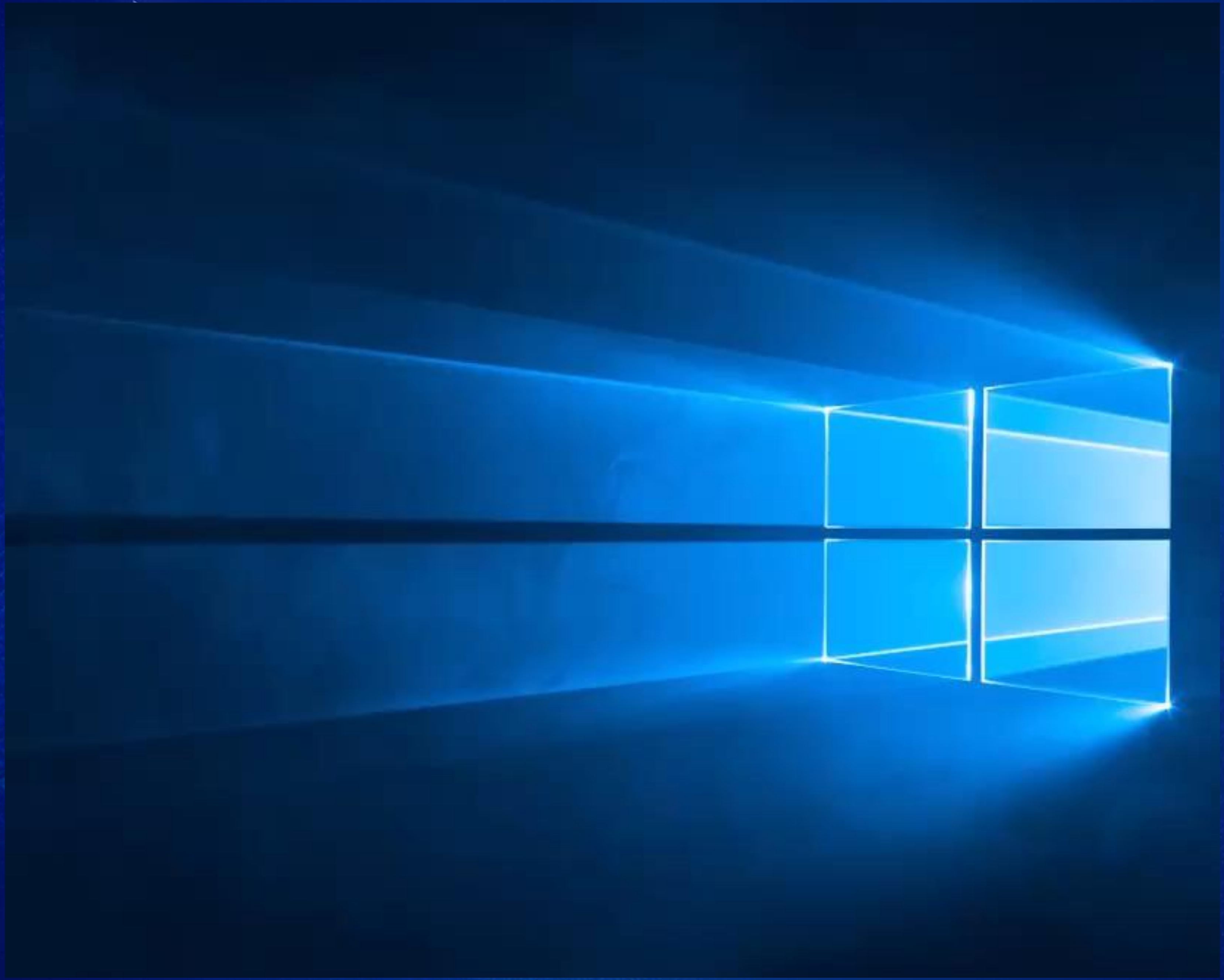
利用技巧

为什么 Explicit Suppression 没起作用?

- Explicit Suppression 清除 NtContinue 在 CFG Bitmap 中的位
- 这里利用的是 NtContinue 之前的函数在 CFG Bitmap 中的位

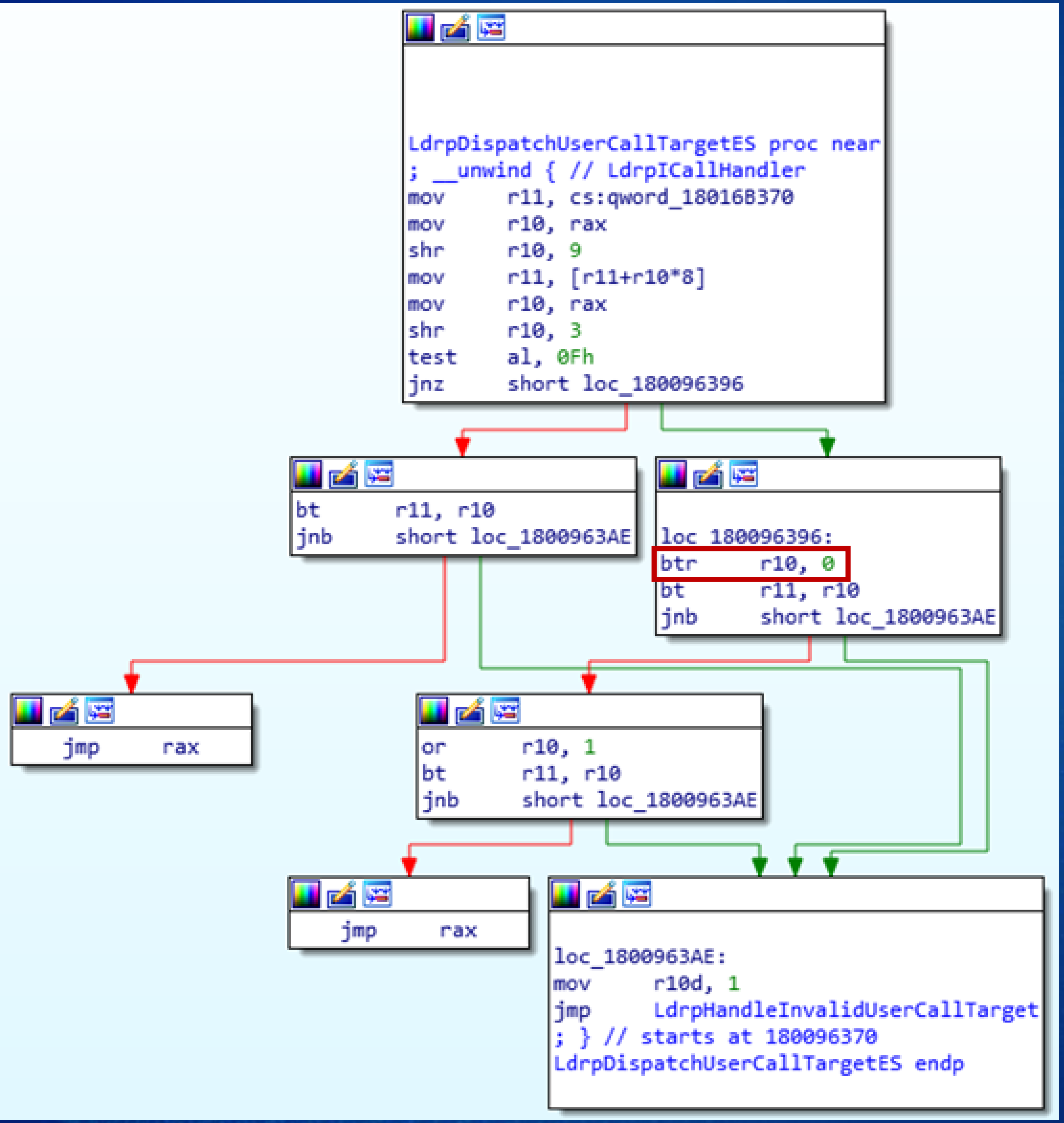
```
76543210 76543210 76543210 76543210
-------- -------- -------- --------
00100010 00010010 00100010 00100000   22 12 22 20
00010010 00100010 00100010 00000010   12 22 22 02
```

```
ntdll!NtDuplicateToken:
00007ffd`2d20a7c0 4c8bd1          mov     r10,rcx
00007ffd`2d20a7c3 b842000000      mov     eax,42h
00007ffd`2d20a7c8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffd`2d20a7d0 7503            jne     ntdll!NtDuplicateToken+0x15 (00007ffd`2d20a7d5)
00007ffd`2d20a7d2 0f05            syscall
00007ffd`2d20a7d4 c3              ret
00007ffd`2d20a7d5 cd2e            int     2Eh
00007ffd`2d20a7d7 c3              ret
00007ffd`2d20a7d8 0f1f840000000000 nop     dword ptr [rax+rax]
ntdll!NtContinue:
00007ffd`2d20a7e0 4c8bd1          mov     r10,rcx
00007ffd`2d20a7e3 b843000000      mov     eax,43h
00007ffd`2d20a7e8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffd`2d20a7f0 7503            jne     ntdll!NtContinue+0x15 (00007ffd`2d20a7f5)
00007ffd`2d20a7f2 0f05            syscall
00007ffd`2d20a7f4 c3              ret
00007ffd`2d20a7f5 cd2e            int     2Eh
00007ffd`2d20a7f7 c3              ret
00007ffd`2d20a7f8 0f1f840000000000 nop     dword ptr [rax+rax]
```



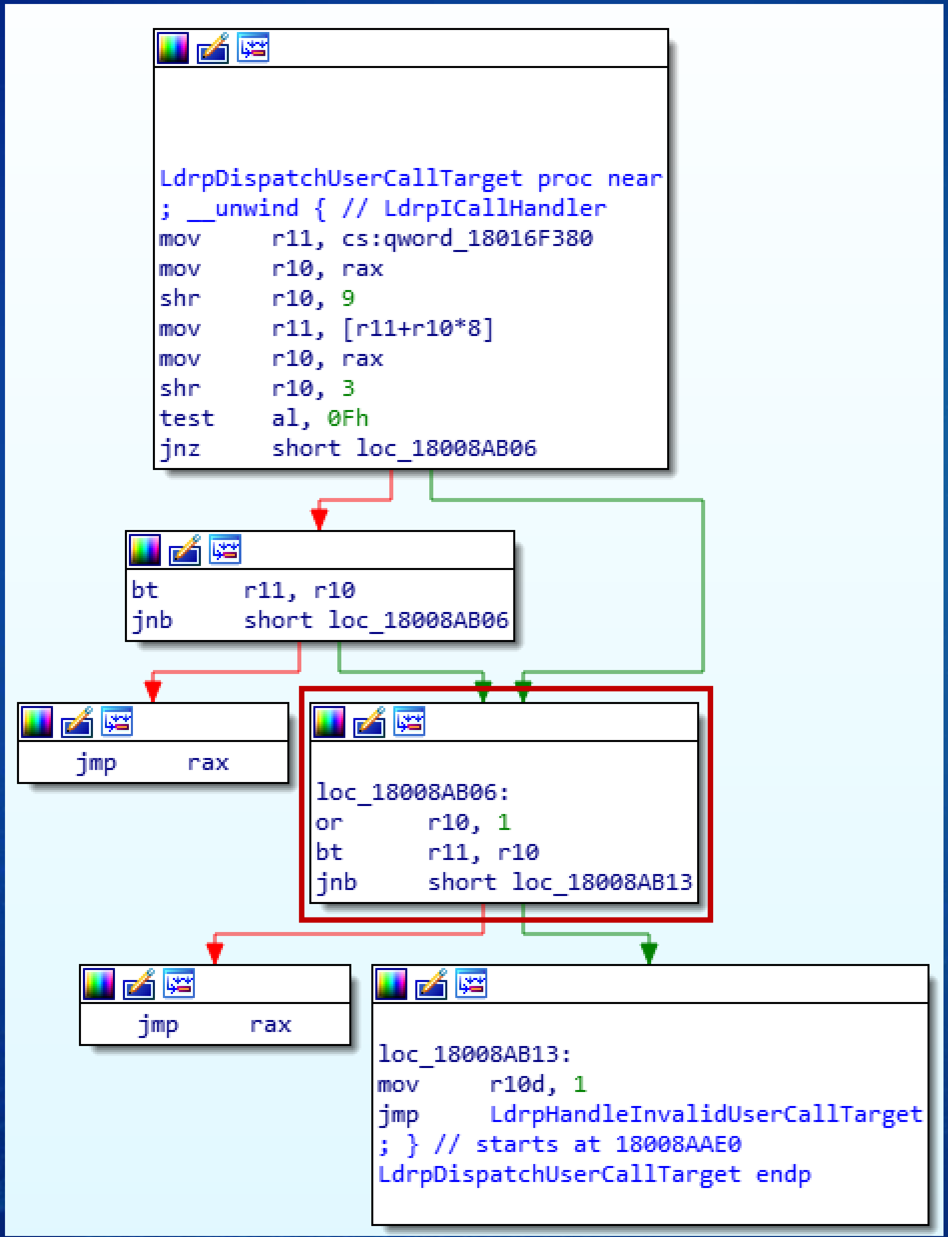
问题修复

Microsoft 在2017年9月修复了这一问题


LdrpDispatchUserCallTargetES proc near
; __unwind { // LdrpICallHandler
mov r11, cs:qword_18016B370
mov r10, rax
shr r10, 9
mov r11, [r11+r10*8]
mov r10, rax
shr r10, 3
test al, 0Fh
jnz short loc_180096396
bt r11, r10
jnb short loc_1800963AE
loc_180096396:
btr r10, 0
bt r11, r10
jnb short loc_1800963AE
jmp rax
or r10, 1
bt r11, r10
jnb short loc_1800963AE
jmp rax
loc_1800963AE:
mov r10d, 1
jmp LdrpHandleInvalidUserCallTarget
; } // starts at 180096370
LdrpDispatchUserCallTargetES endp

并没有完全解决问题

LdrpDispatchUserCallTarget 仍然使用着错误的逻辑


LdrpDispatchUserCallTarget proc near
; __unwind { // LdrpICallHandler
mov r11, cs:qword_18016F380
mov r10, rax
shr r10, 9
mov r11, [r11+r10*8]
mov r10, rax
shr r10, 3
test al, 0Fh
jnz short loc_18008AB06
bt r11, r10
jnb short loc_18008AB06
jmp rax
loc_18008AB06:
or r10, 1
bt r11, r10
jnb short loc_18008AB13
jmp rax
loc_18008AB13:
mov r10d, 1
jmp LdrpHandleInvalidUserCallTarget
; } // starts at 18008AAE0
LdrpDispatchUserCallTarget endp

利用技巧

加载时：根据 GuardFlags 更新函数指针__guard_dispatch_icall_fptr

```
if ( !LdrControlFlowGuardEnforcedWithExportSuppression()
  || (fptr = LdrpDispatchUserCallTargetES, !(*(LoadConfig + 0x90) & 0x4000)) )
{
  fptr = LdrpDispatchUserCallTarget;
}
*_guard_dispatch_icall_fptr = fptr;
```

IMAGE_GUARD_CF_EXPORT_SUPPRESSION_INFO_PRESENT

利用技巧

没有设置该标志位的动态链接库

- C:\Windows\System32\F12\msdbg2.dll
- C:\Windows\System32\F12\pdm.dll
- C:\Windows\System32\F12\pdmproxy100.dll
- C:\Windows\System32\Macromed\Flash\Flash.ocx
- C:\Windows\System32\Macromed\Flash\FlashUtil_ActiveX.dll
- C:\Windows\System32\aspnet_counters.dll
- C:\Windows\System32\libcrypto.dll
- ……

利用技巧

寻找一个跳板函数

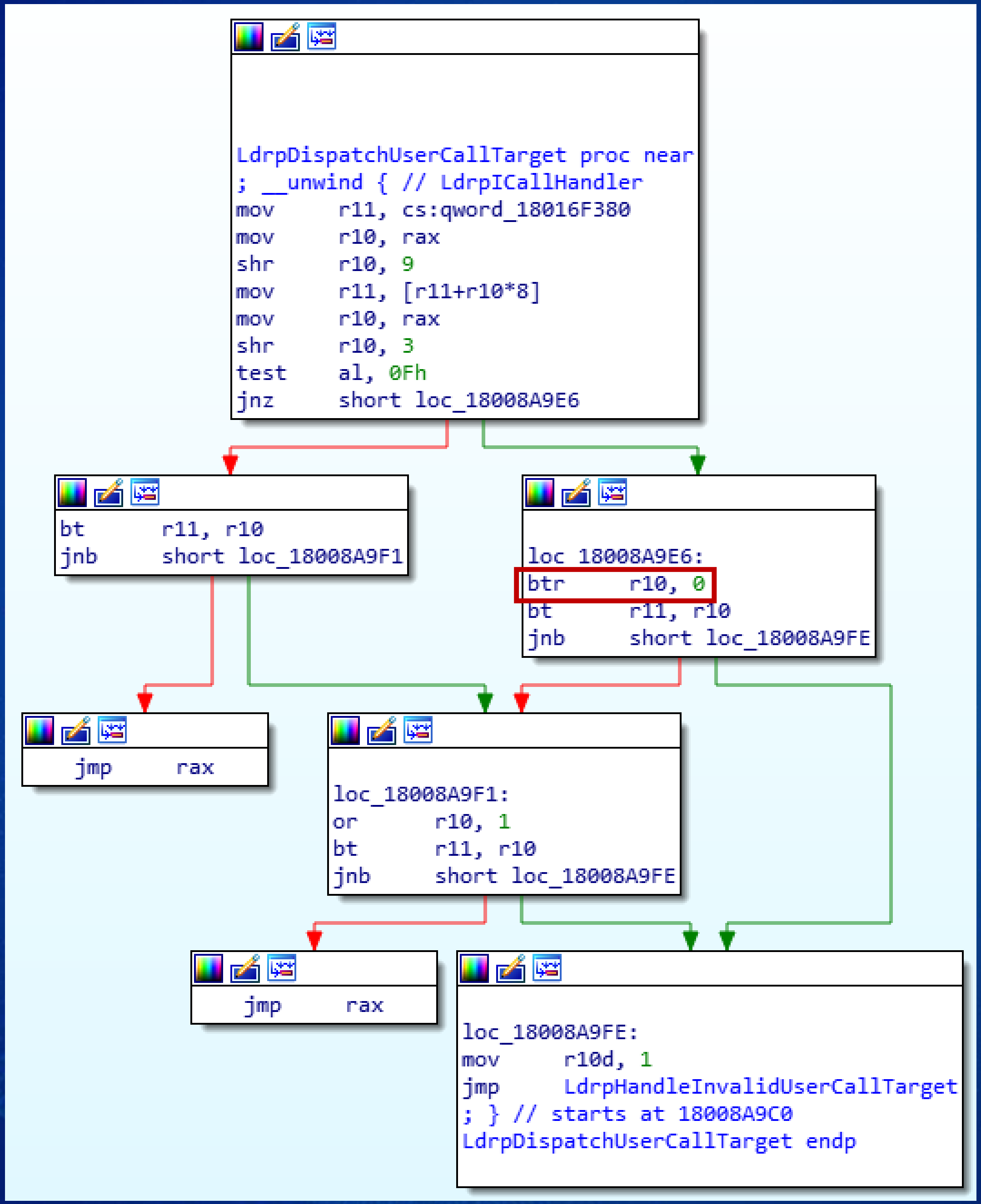
- 函数本身可以被间接调用
- 函数内会调用另一函数（地址与参数均可控）

```
; __int64 __fastcall CApplicationThread::onDestroyThreadComplete(CApplicationThread *__hidden this)
public: virtual long CApplicationThread::onDestroyThreadComplete(void) proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h

mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rsi
push    rdi
sub     rsp, 20h
mov     rdi, rcx
mov     dword ptr [rcx+88h], 10h
lock inc dword ptr [rcx+28h]
cmp     qword ptr [rcx+98h], 0
jz      short loc_180006450
```

```
lea     rsi, [rcx-20h]
mov     rax, [rsi]
mov     rbx, [rax+70h]
mov     rcx, rbx        ; this
call    cs:__guard_check_icall_fptr
mov     rcx, rsi
call    rbx
cmp     eax, 1
jnz     short loc_180006427
```

再次修复

Microsoft 在2018年6月再次修复了这一问题


LdrpDispatchUserCallTarget proc near
; __unwind { // LdrpICallHandler
mov r11, cs:qword_18016F380
mov r10, rax
shr r10, 9
mov r11, [r11+r10*8]
mov r10, rax
shr r10, 3
test al, 0Fh
jnz short loc_18008A9E6
bt r11, r10
jnb short loc_18008A9F1
loc_18008A9E6:
btr r10, 0
bt r11, r10
jnb short loc_18008A9FE
jmp rax
loc_18008A9F1:
or r10, 1
bt r11, r10
jnb short loc_18008A9FE
jmp rax
loc_18008A9FE:
mov r10d, 1
jmp LdrpHandleInvalidUserCallTarget
; } // starts at 18008A9C0
LdrpDispatchUserCallTarget endp

THANKS