The Joy of Sandbox Mitigations

James Forshaw @tiraniddo
Troopers 2016

What I’m Going to Talk About

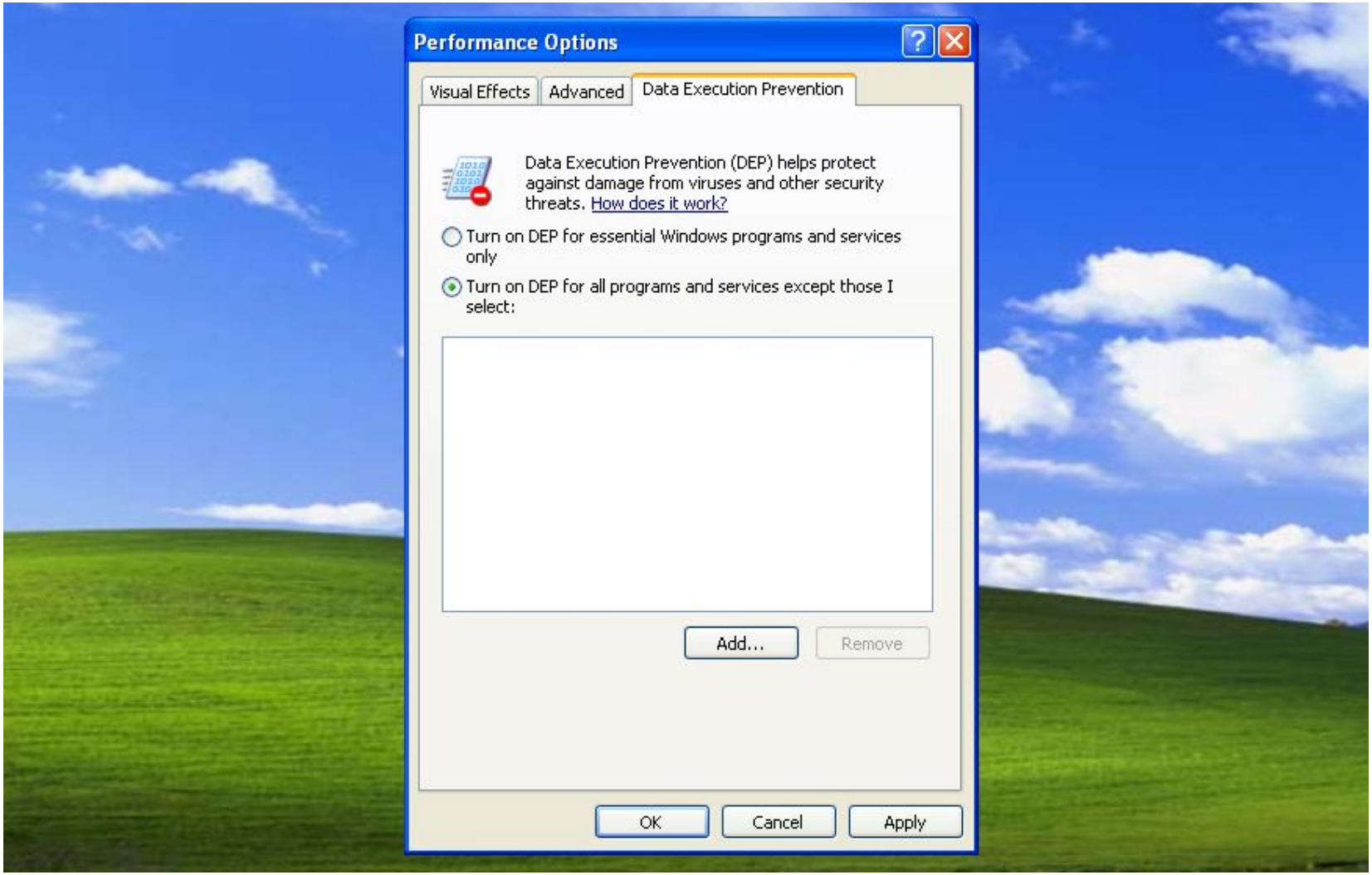
- Sandbox related mitigations added to Windows since Windows 8
- Hardcore reverse engineering of the implementation
- What’s supported and how you can use them yourself
- What is blocked if you’re investigating sandbox attack surface
- Weird bugs and edge cases with the implementation
- Research on Windows 8.1 and 10 build 10586

WARNING

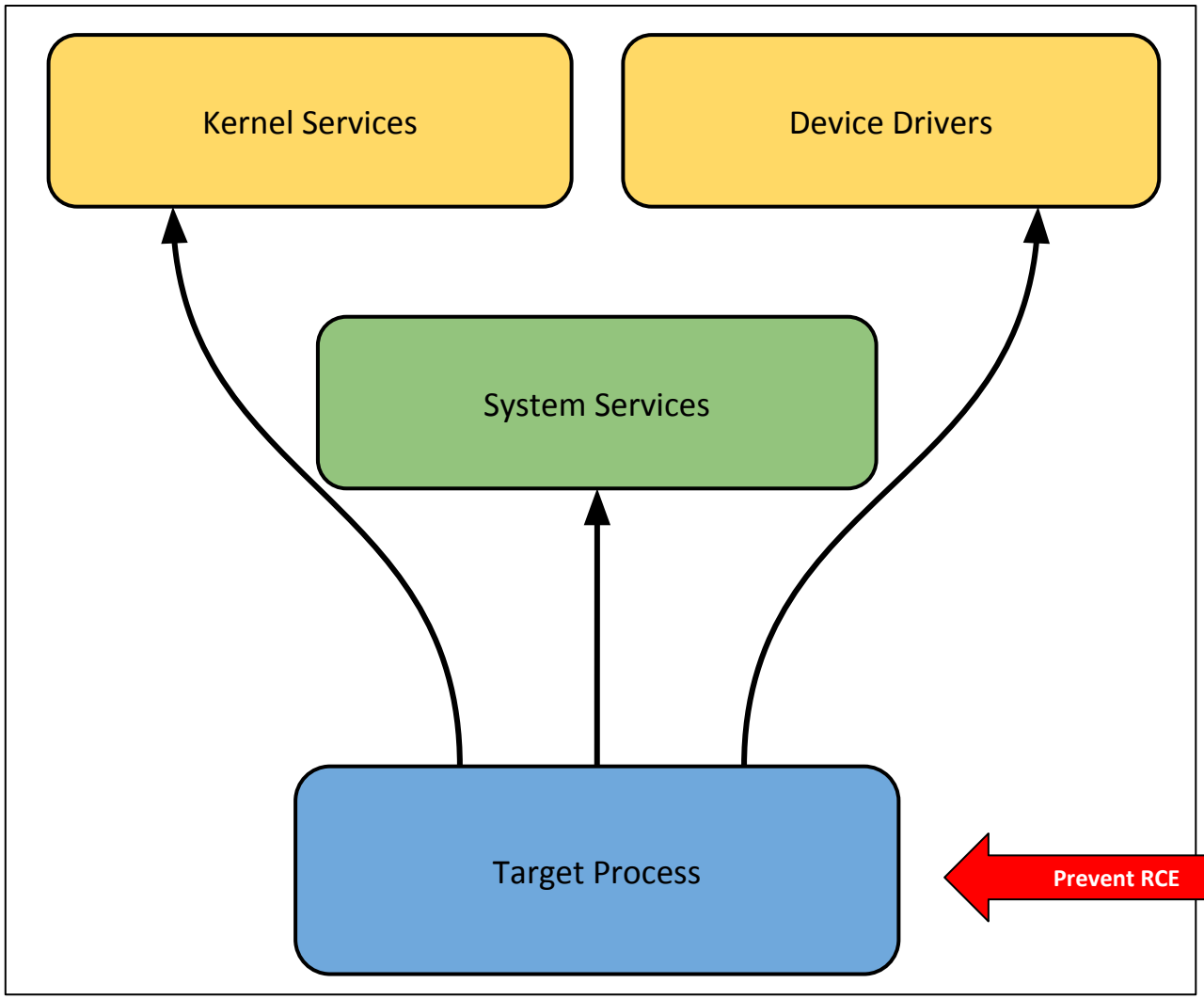
A huge amount of code snippets is approaching.

Sorry?

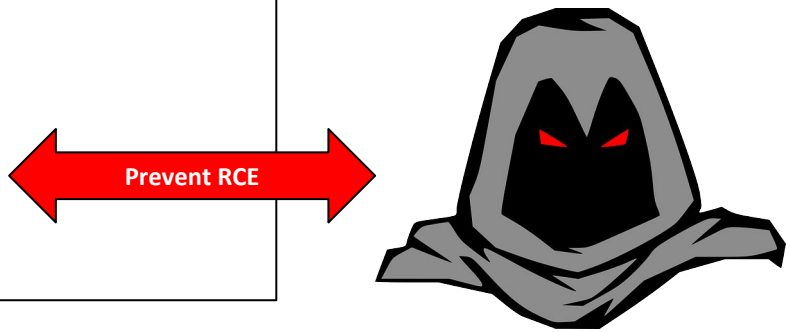
In the Beginning was DEP



The External Attacker's Viewpoint



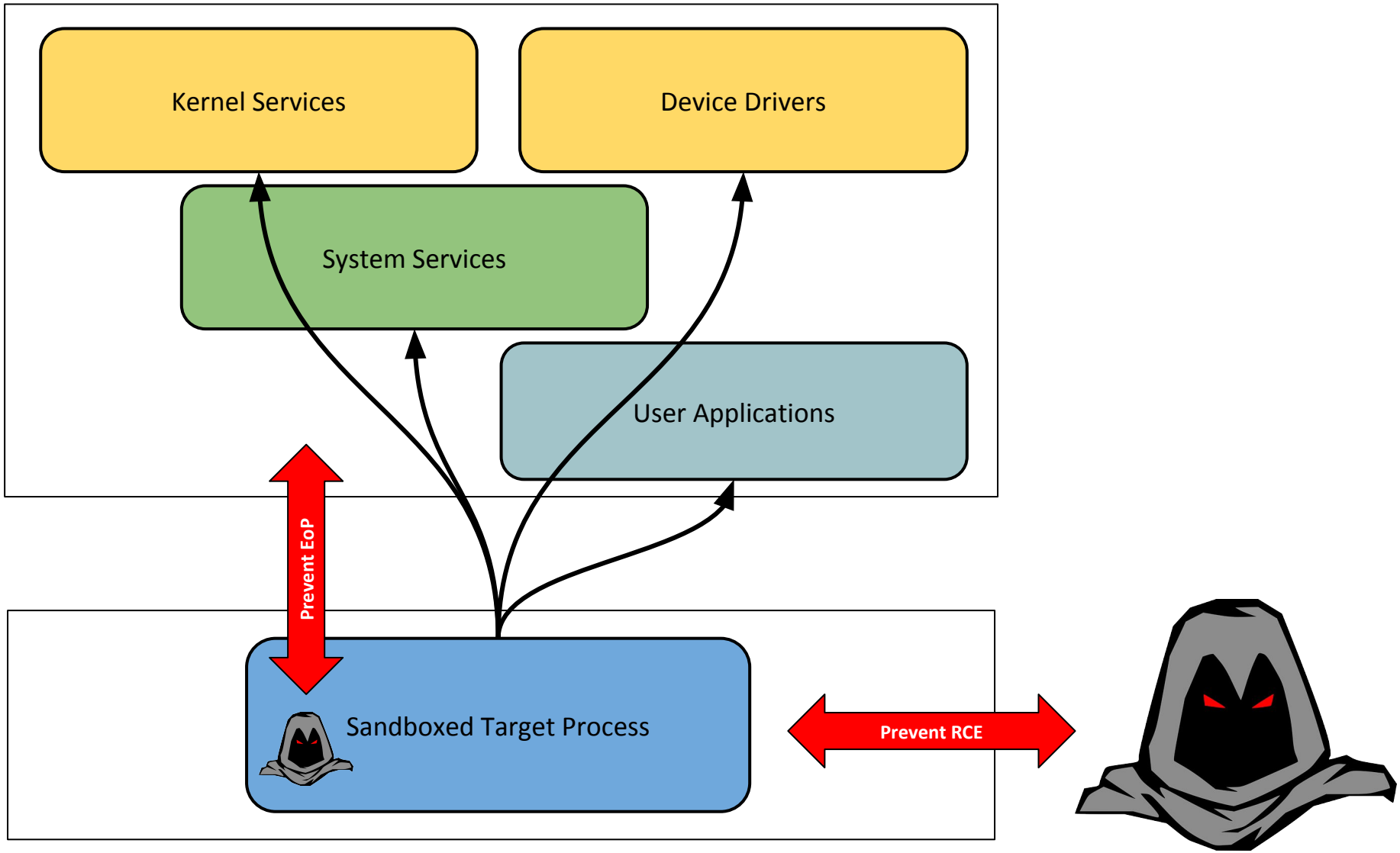
Mitigations:
DEP
ASLR
SafeSEH
Stack Cookies
Control Flow Guard (CFG)
etc.



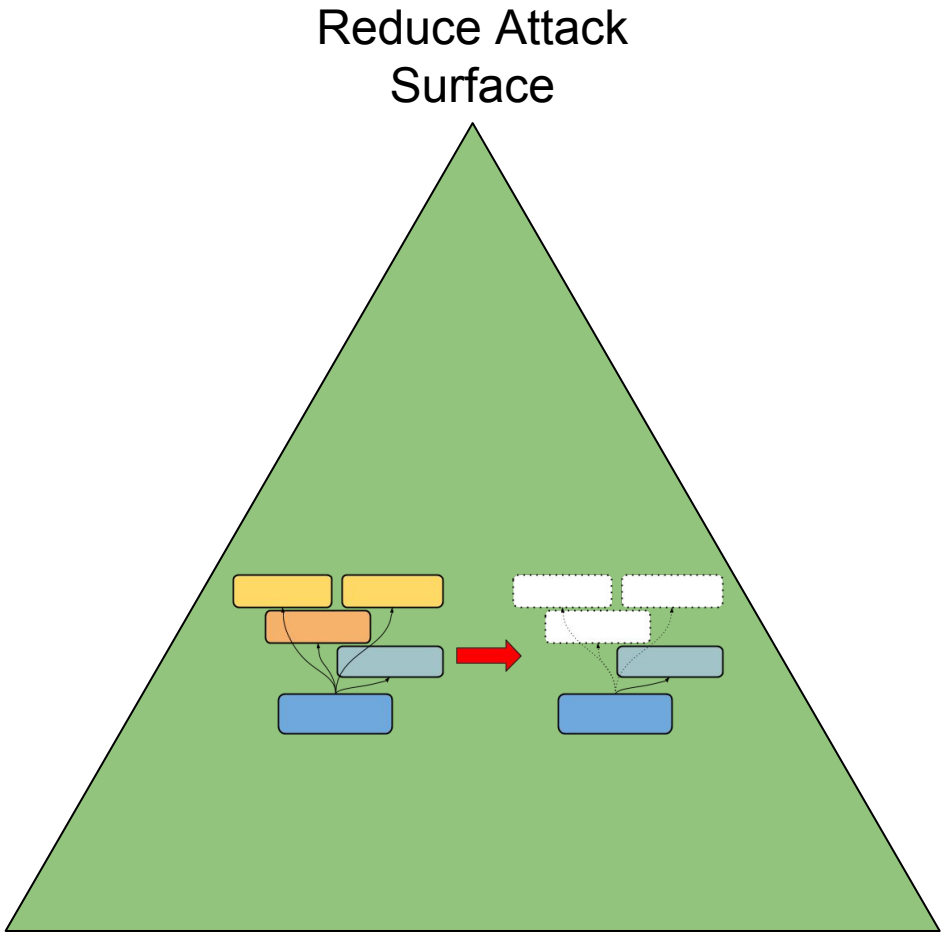
The Defender's Dilemma

Assume Compromise

Sandboxing to the Rescue?



Technical Mitigations

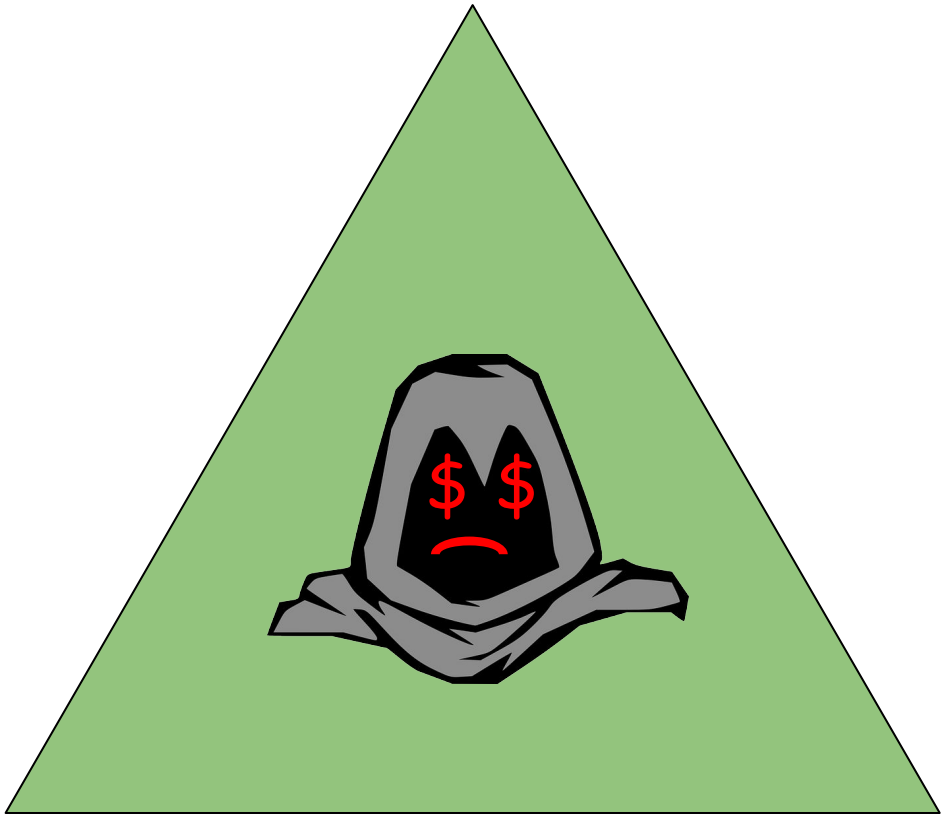


Reduce Exploit
Reliability

Increase Exploit
Cost

Technical Mitigations

Reduce Attack Surface

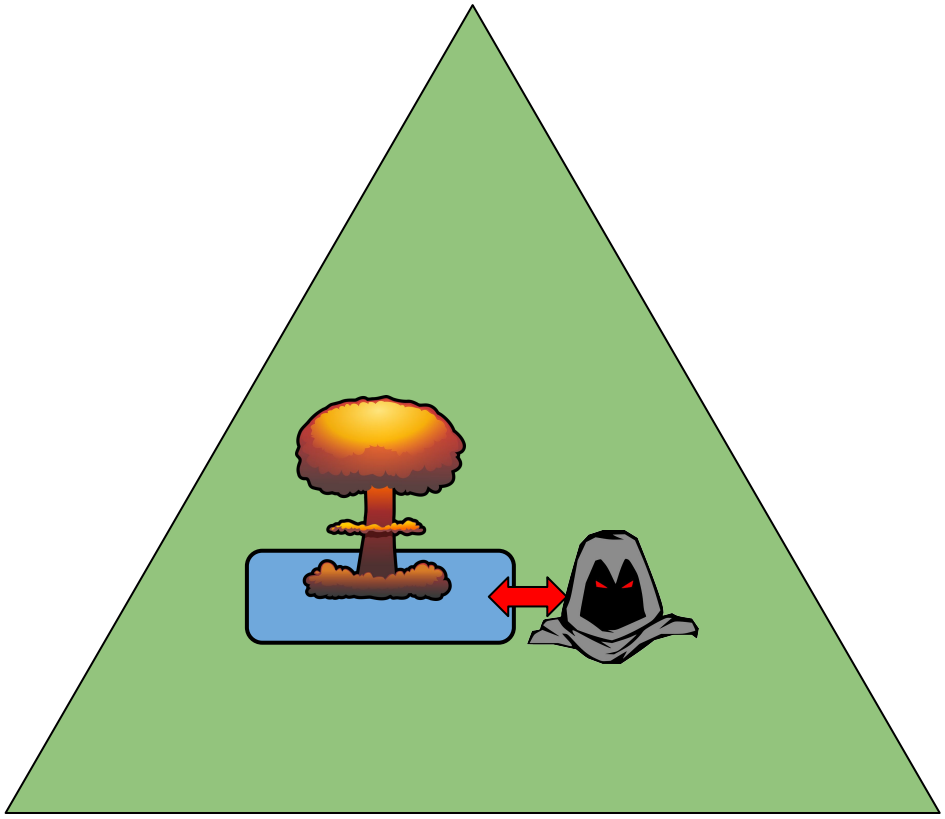


Reduce Exploit Reliability

Increase Exploit Cost

Technical Mitigations

Reduce Attack Surface



Reduce Exploit Reliability

Increase Exploit Cost

Categories of Windows Sandbox Mitigations

Explicit Mitigations

- Mitigations which must be actively enabled
- Usually more disruptive, requires changes to existing code to support
- Aim to reduce attack surface or increase exploitation costs

Implicit Mitigations

- On by default when running sandboxed code
- Applies to select features which shouldn't affect typical backwards compatibility support
- Focus more on breaking chains and making exploitation harder

Explicit Mitigations

Setting an Explicit Mitigation Policy

SetProcessMitigationPolicy function

Sets the mitigation policy for the calling process.

Syntax

C++

```
BOOL WINAPI SetProcessMitigationPolicy(
  _In_ PROCESS_MITIGATION_POLICY MitigationPolicy,
  _In_ PVOID                     lpBuffer,
  _In_ SIZE_T                    dwLength
);
```

Policy Type

Accompanying Data

PROCESS_MITIGATION_POLICY

Policy	*Supported Win8.1 Update 2*	*Supported Win10 TH2*
ProcessDEPPolicy	Yes	Yes
ProcessASLRPolicy	Yes	Yes
ProcessDynamicCodePolicy	Yes	Yes
ProcessStrictHandleCheckPolicy	Yes	Yes
ProcessSystemCallDisablePolicy	Yes	Yes
ProcessMitigationOptionsMask	Invalid	Invalid
ProcessExtensionPointDisablePolicy	Yes	Yes
ProcessControlFlowGuardPolicy	Invalid	Invalid
ProcessSignaturePolicy	Yes*	Yes
ProcessFontDisablePolicy	No	Yes
ProcessImageLoadPolicy	No	Yes

* Not supported through SetProcessMitigationPolicy

PROCESS_MITIGATION_POLICY

Policy	*Supported Win8.1 Update 2*	*Supported Win10 TH2*
ProcessDEPPolicy	Yes	Yes
ProcessASLRPolicy	Yes	Yes
ProcessDynamicCodePolicy	Yes	Yes
ProcessStrictHandleCheckPolicy	Yes	Yes
ProcessSystemCallDisablePolicy	Yes	Yes
ProcessMitigationOptionsMask	Invalid	Invalid
ProcessExtensionPointDisablePolicy	Yes	Yes
ProcessControlFlowGuardPolicy	Invalid	Invalid
ProcessSignaturePolicy	Yes*	Yes
ProcessFontDisablePolicy	No	Yes
ProcessImageLoadPolicy	No	Yes

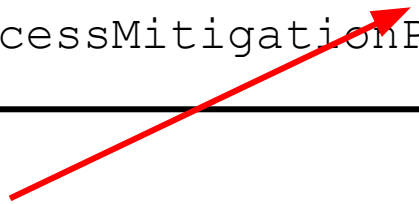
* Not supported through SetProcessMitigationPolicy

Under the Hood

```
const int ProcessMitigationPolicy = 52;

struct PROCESS_MITIGATION {
  PROCESS_MITIGATION_POLICY Policy;
  DWORD Flags;
};

PROCESS_MITIGATION m = {ProcessSignaturePolicy, 1};

NtSetInformationProcess(GetCurrentProcess(),
            ProcessMitigationPolicy, &m, sizeof(m));
```

Under the Hood

```
const int ProcessMitigationPolicy = 52;

struct PROCESS_MITIGATION {
  PROCESS_MITIGATION_POLICY Policy;
  DWORD Flags;
};

PROCESS_MITIGATION m = {ProcessSignaturePolicy, 1};

NtSetInformationProcess(GetCurrentProcess(),
            ProcessMitigationPolicy, &m, sizeof(m));
```



Can’t specify anything other than current process, it will fail.

Apply During Create Process

Process Attribute for Mitigation Policy

```
LPPROC_THREAD_ATTRIBUTE_LIST attr_list = ... // Allocated
InitializeProcThreadAttributeList(attr_list, 1, 0, &size);

DWORD64 policy = PROCESS_CREATION_MITIGATION_POLICY_...;
UpdateProcThreadAttribute(attr_list, 0,
    PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY, &policy,
    sizeof(policy), nullptr, nullptr);

startInfo.StartupInfo.cb = sizeof(startInfo);
startInfo.lpAttributeList = attr_list;

// Create Process with EXTENDED_STARTUPINFO_PRESENT flag
```

I'll abbreviate PROCESS_CREATION_MITIGATION_POLICY to PCMP otherwise it gets too long

Image File Execution Options

- Specify REG_QWORD value *MitigationOptions* with same bit fields as used at process creation.

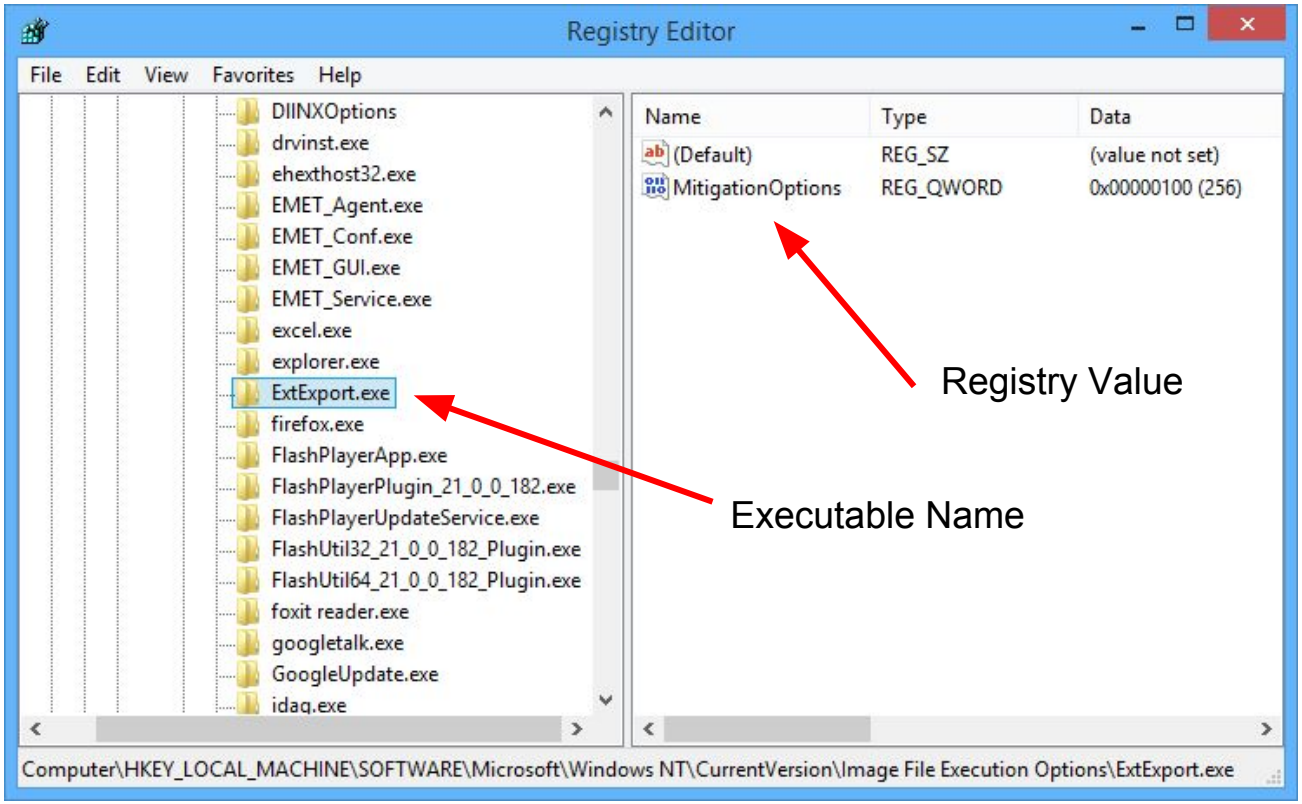
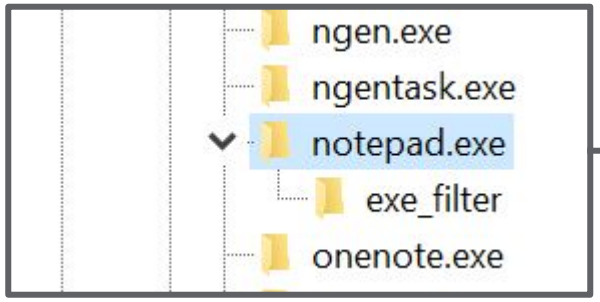
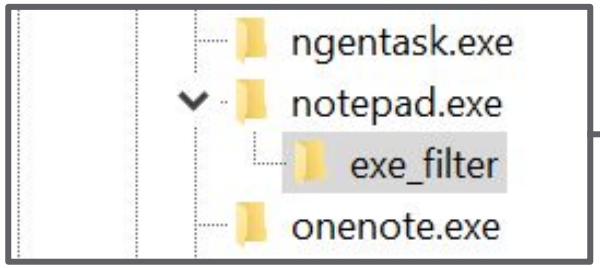


Image File Execution Options Filter



Name	Type	Data
(Default)	REG_SZ	(value not set)
UseFilter	REG_DWOR...	0x00000001 (1)



Name	Type	Data
(Default)	REG_SZ	(value not set)
FilterFullPath	REG_SZ	c:\windows\notepad.exe
MitigationOptions	REG_QWO...	0x101000000000 (17660

Demo Time!

Simple Mitigation Tests

Dynamic Code Policy

Category: Increase Exploit Cost, Reduce Reliability

```
struct PROCESS_MITIGATION_DYNAMIC_CODE_POLICY {
    union {
        DWORD Flags;
        struct {
            DWORD ProhibitDynamicCode : 1;
            DWORD ReservedFlags : 31;
        };
    };
}

PCMP_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON   (1 << 36)
```

What Does it Disable?

- Calling VirtualAlloc with PAGE_EXECUTE_*
- MapViewOfFile with FILE_MAP_EXECUTE option
- Calling VirtualProtect with PAGE_EXECUTE_* etc.
- Reconfiguring the CFG bitmap via SetProcessValidCallTargets

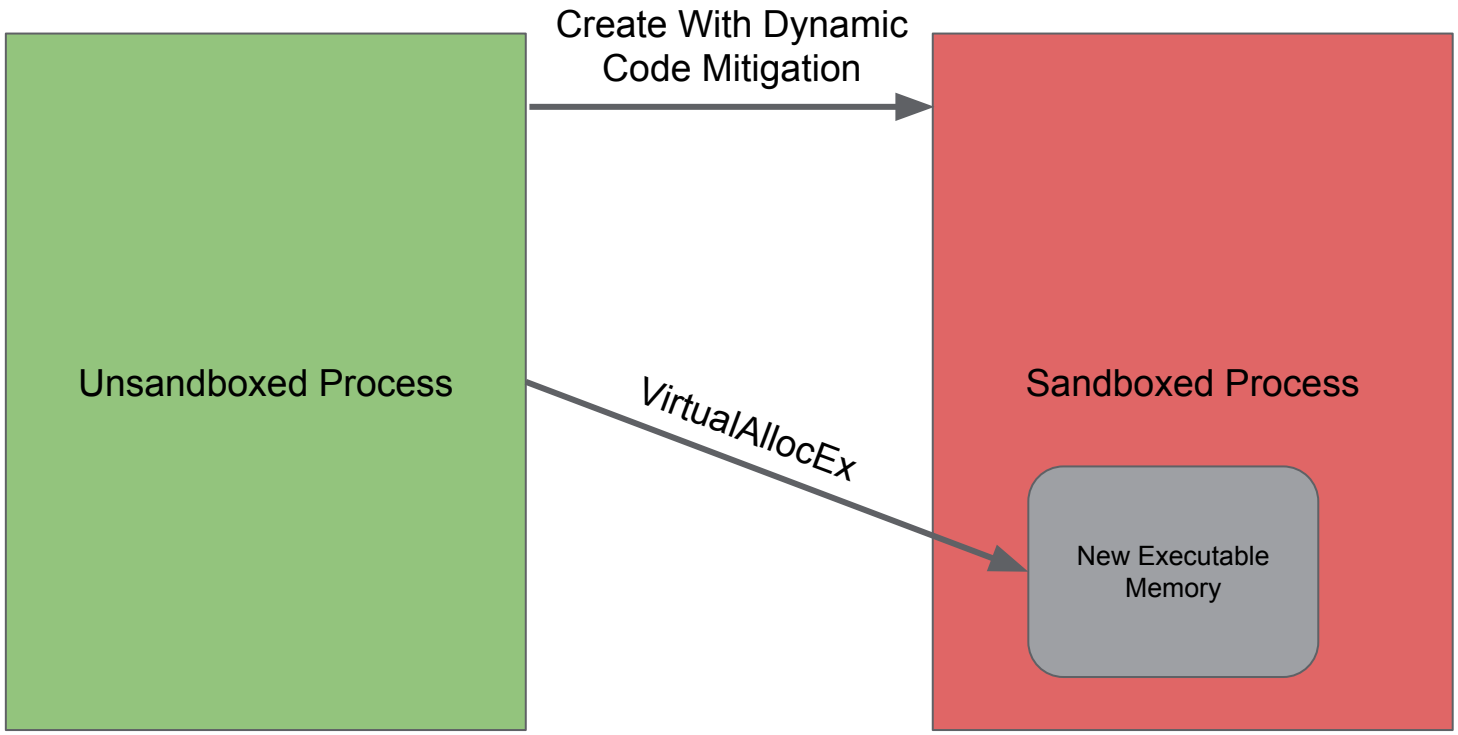
It doesn’t disable loading DLLs/Image Sections!

Under the Hood

```
NTSTATUS MiArbitraryCodeBlocked(EPROCESS *Process) {
  if (Process->Flags2 & DYNAMIC_CODE_BLOCKED_FLAG)
    return STATUS_DYNAMIC_CODE_BLOCKED;
  return STATUS_SUCCESS;
}
```

Might assume the *Process* argument is the target process.
You'd be wrong of course!
It's the calling process.

Fun Use Case

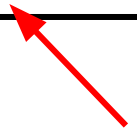



```
void InjectExecutableCode(DWORD dwPid, PBYTE pData, SIZE_T nSize) {
  HANDLE hProcess = OpenProcess(dwPid,
                        PROCESS_VM_WRITE| PROCESS_VM_OPERATION);
  LPVOID pMem = VirtualAllocEx(hProcess,
                        PAGE_EXECUTE_READWRITE, nSize);
  WriteProcessMemory(hProcess, pMem, pData, nSize);
}
```

Binary Signature Policy

Category: Increase Exploit Cost, Reduce Reliability

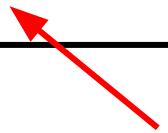
```
struct PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY {
    union {
        DWORD Flags;
        struct {
            DWORD MicrosoftSignedOnly : 1;
            DWORD StoreSignedOnly : 1; // Win10 Only.
            DWORD MitigationOptIn : 1; // Win10 Only.
            DWORD ReservedFlags : 29;
        };
    };
};

PCMP_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON   (1 << 44)
```



Can only specify Microsoft Signed at Process Creation

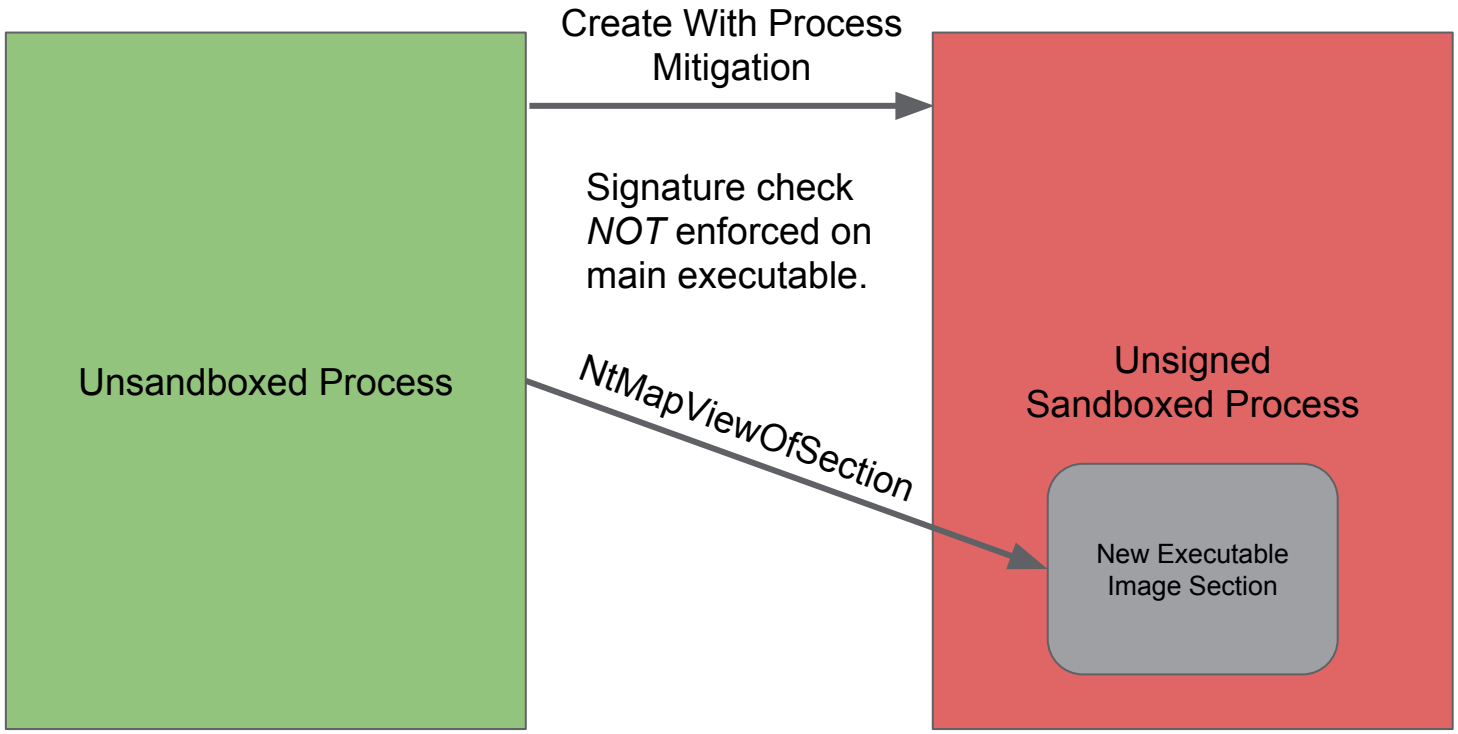
Under the Hood

```
case ProcessSignaturePolicy:
 if (policy.MicrosoftSignedOnly) {
   process->SignatureLevel = 8;
   process->SectionSignatureLevel = 8;
 } else if (policy.StoreSignedOnly) {
   process->SignatureLevel = 6;
   process->SectionSignatureLevel = 6;
 }
 process->Flags2 |= 0x2000;
```



MitigationOptIn is used to set the Flags2 flag, which opts into Store Signing checks if an image is loaded which requires code signing.

Fun Use Case




```
void* InjectExecutableImage(DWORD dwPid, HANDLE hMap) {
  LPVOID lpMap = NULL;
  HANDLE hProcess = OpenProcess(dwPid,
                    PROCESS_VM_WRITE| PROCESS_VM_OPERATION);
  NtMapViewOfSection(hMap, hProcess, &lpMap, 0,
     4096, nullptr, &viewsize,
     ViewUnmap, 0, PAGE_EXECUTE_READWRITE);
  return lpMap;
}
```

NtMapViewOfSection can take a process handle

Image Load Policy (Win10 Only)

Category: Increase Exploit Cost, Reduce Reliability

```
struct PROCESS_MITIGATION_IMAGE_LOAD_POLICY {
    union {
        DWORD Flags;
        struct {
            DWORD NoRemoteImages : 1;
            DWORD NoLowMandatoryLabelImages : 1;
            DWORD ReservedFlags : 30;
        };
    };
};

PCMP_IMAGE_LOAD_NO_REMOTE_ALWAYS_ON     (1 << 52)
PCMP_IMAGE_LOAD_NO_LOW_LABEL_ALWAYS_ON  (1 << 56)
```

Under the Hood

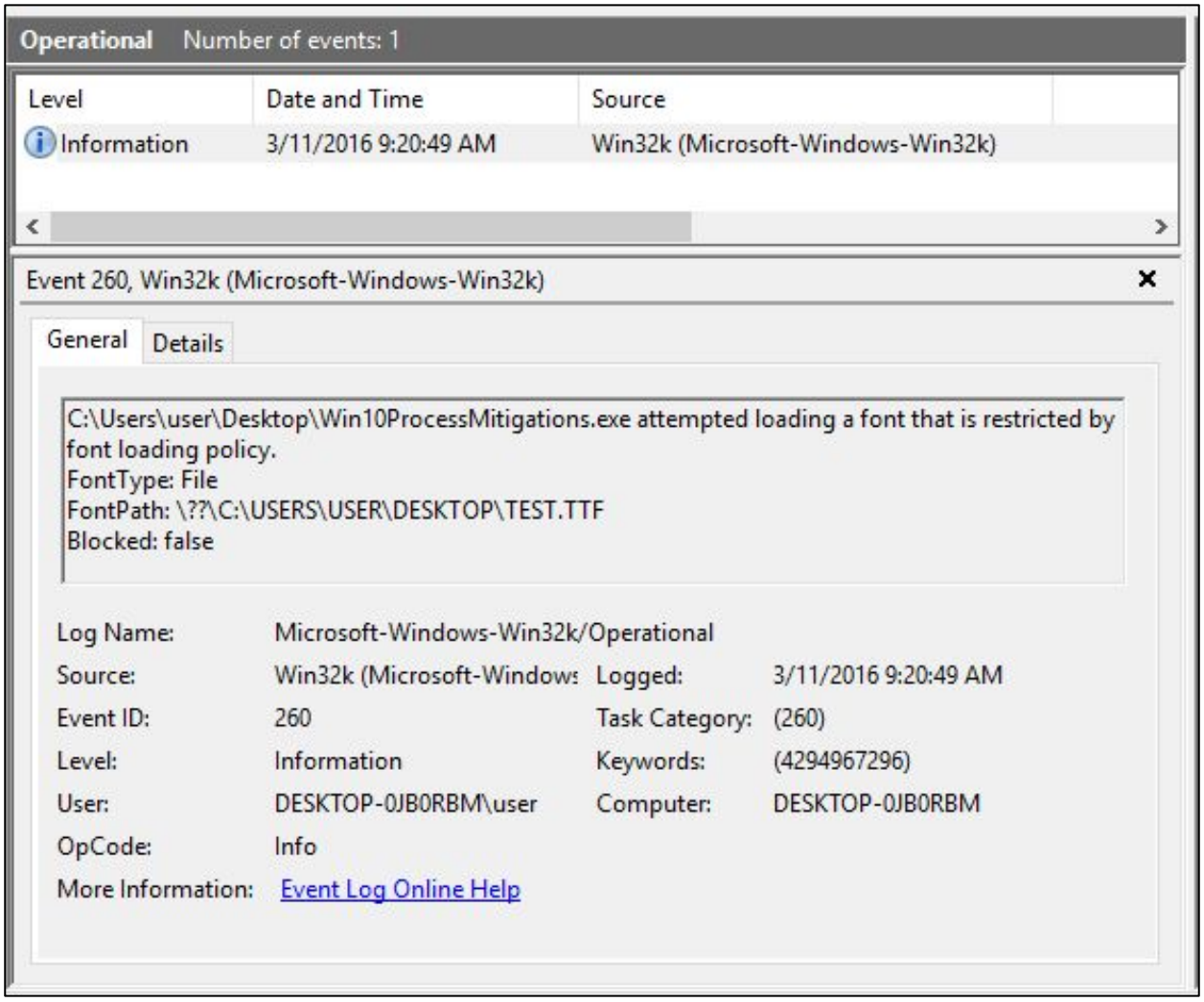
```
NTSTATUS MiAllowImageMap(EPROCESS *process, SECTION*
section)
{
  unsigned int flags3 = process->Flags3;
  if (flags3 & PROCESS_FLAGS3_NOREMOTEIMAGE &&
      (section->FileObject->RemoteImageFileObject ||
       section->FileObject->RemoteDataFileObject))
    return STATUS_ACCESS_DENIED;

  if (flags3 & PROCESS_FLAGS3_NOLOWILIMAGE) {
    DWORD il = 0;
    SeQueryObjectMandatoryLabel(section->FileObject, &il);
    if (il <= SECURITY_MANDATORY_LOW_RID)
      return STATUS_ACCESS_DENIED;
  }
  return STATUS_SUCCESS;
}
```

Font Disable Policy (Win10 Only)

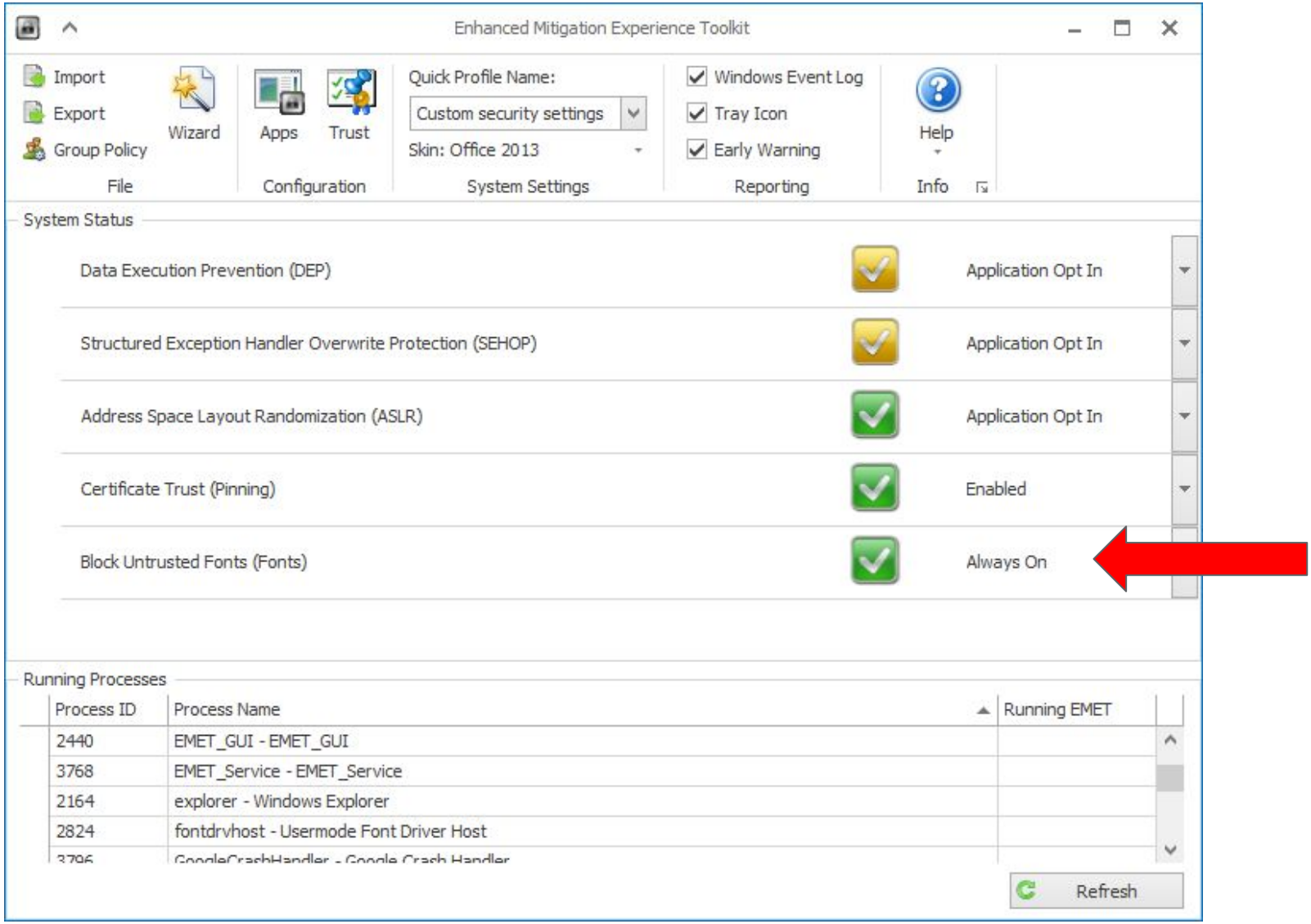
Category: Reduced Attack Surface

```
struct PROCESS_MITIGATION_FONT_DISABLE_POLICY {
    union {
        DWORD Flags;
        struct {
            DWORD DisableNonSystemFonts      : 1;
            DWORD AuditNonSystemFontLoading : 1;
            DWORD ReservedFlags              : 30;
        };
    };
};

PCMP_FONT_DISABLE_ALWAYS_ON (1 << 48)
PCMP_AUDIT_NONSYSTEM_FONTS  (3 << 48)
```

Font Disable Auditing



Also Available in EMET 5.5



Under the Hood

- Win32k shouldn't have insider knowledge of EPROCESS Flags

```
int GetCurrentProcessFontLoadingOption()
{
  PROCESS_MITIGATION_FONT_DISABLE_POLICY policy;

  ZwQueryInformationProcess((HANDLE)-1,
     ProcessMitigationPolicy, &policy, sizeof(policy), 0);
  if (policy.DisableNonSystemFonts)
    return 2;
  else
    return policy.AuditNonSystemFontLoading;
}
```

How it Works

```
int WIN32K::bLoadFont(...) {
  int load_option = GetCurrentProcessFontLoadingOption();
  bool system_font = true;
  if (load_option) {
    HANDLE hFile = hGetHandleFromFilePath(FontPath);
    BOOL system_font = bIsFileInSystemFontsDir(hFile);
    ZwClose(hFile);
    if (!system_font) {
      LogFontLoadAttempt(FontPath);
      if (load_option == 2)
        return 0;
    }
  }
  HANDLE hFont = hGetHandleFromFilePath(FontPath);
  // Map font as section
}
```

Demo Time!

Bypass Font Path Check

Syscall Disable Policy

Category: Reduced Attack Surface

```
struct PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY {
    union {
        DWORD Flags;
        struct {
            DWORD DisallowWin32kSystemCalls : 1;
            DWORD ReservedFlags : 31;
        };
    };
};

PCMP_WIN32K_SYSTEM_CALL_DISABLE_ALWAYS_ON (1 << 28)
```

Win32k Syscall Disable Implementation?

```
SyscallEnter() {
  // Setup entry.
  if (IsWin32kSyscall(SyscallNumber) &&
      CurrentProcess->Flags2 & WIN32K_SYSCALL_DISABLE) {
    return STATUS_ACCESS_DENIED;
  }

  // Continue normal system call dispatch.
}
```

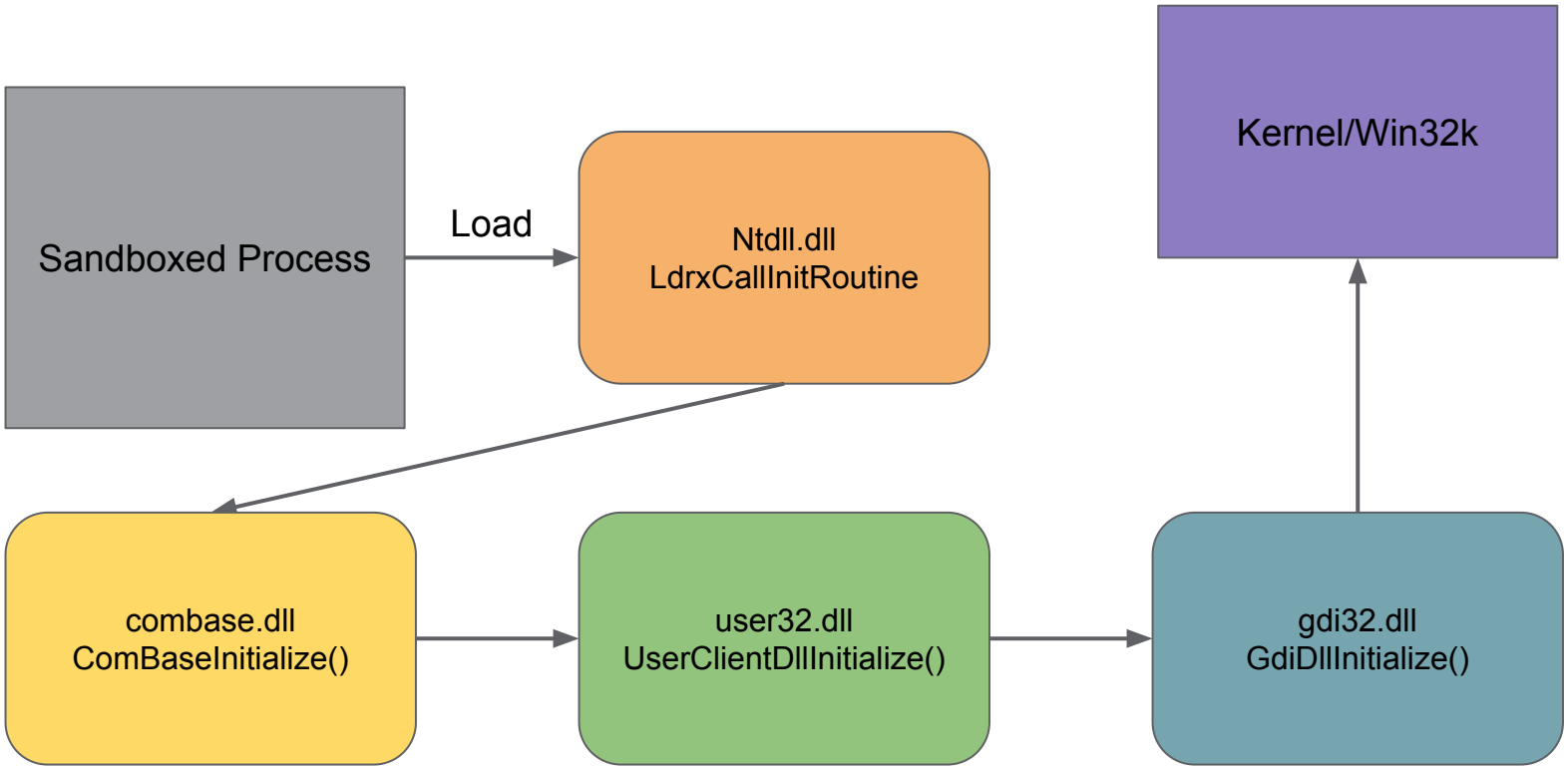
Win32k Syscall Disable

```
SyscallEnter() {
  SERVICE_TABLE* table =
                KeGetCurrentThread()->ServiceTable;

  // Setup entry.
  if (IsWin32Syscall(SyscallNo) &&
      SyscallNo & 0xFFF > table->Shadow.Length)
    if (PsConvertToGuiThread() != STATUS_SUCCESS)
      return GetWin32kResult(SyscallNo);
  }

  // Continue normal system call dispatch.
}
```

Win32k Syscall Disable is Per Thread

```
NTSTATUS PsConvertToGuiThread() {
  KTHREAD *thread = KeGetCurrentThread();
  EPROCESS *process = thread->Process;

  if (thread->ServiceTable != &KeServiceDescriptorTable)
    return STATUS_ALREADY_WIN32;
  if (process->Flags2 & WIN32K_SYSCALL_DISABLE)
    return STATUS_ACCESS_DENIED;

  thread->ServiceTable = &KeServiceDescriptorTableShadow;
  NTSTATUS result = PsInvokeWin32Callout(THREAD_INIT,
                                         &thread);
  if (result < 0)
    thread->ServiceTable = &KeServiceDescriptorTable;
  return result;
}
```

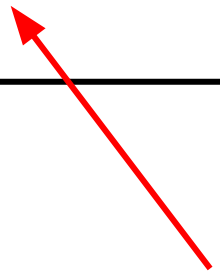
Late Enable Initial Thread Problem



Initial Thread now a GUI Thread!

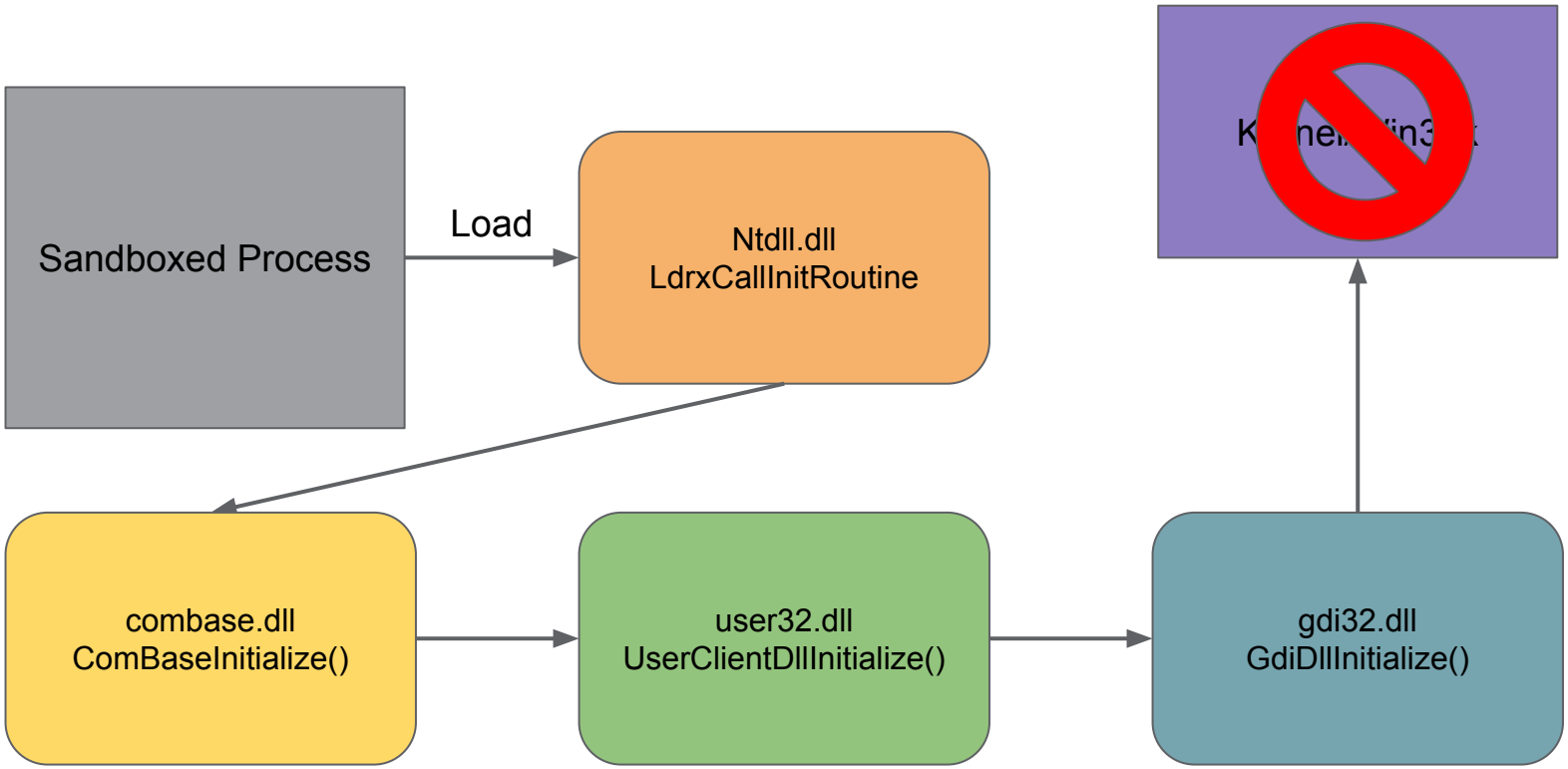
Disable Dynamically

```
case ProcessSystemCallDisablePolicy:
  PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY policy;
  if (!policy.DisallowWin32kSystemCalls
      && process->Flags2 & WIN32K_SYSCALL_DISABLE) {
    return STATUS_ACCESS_DENIED;
  }
  process->Flags2 |= WIN32K_SYSCALL_DISABLE;
  if (thread->ServiceTable != &KeServiceDescriptorTable) {
    return STATUS_TOO_LATE;
  }
```

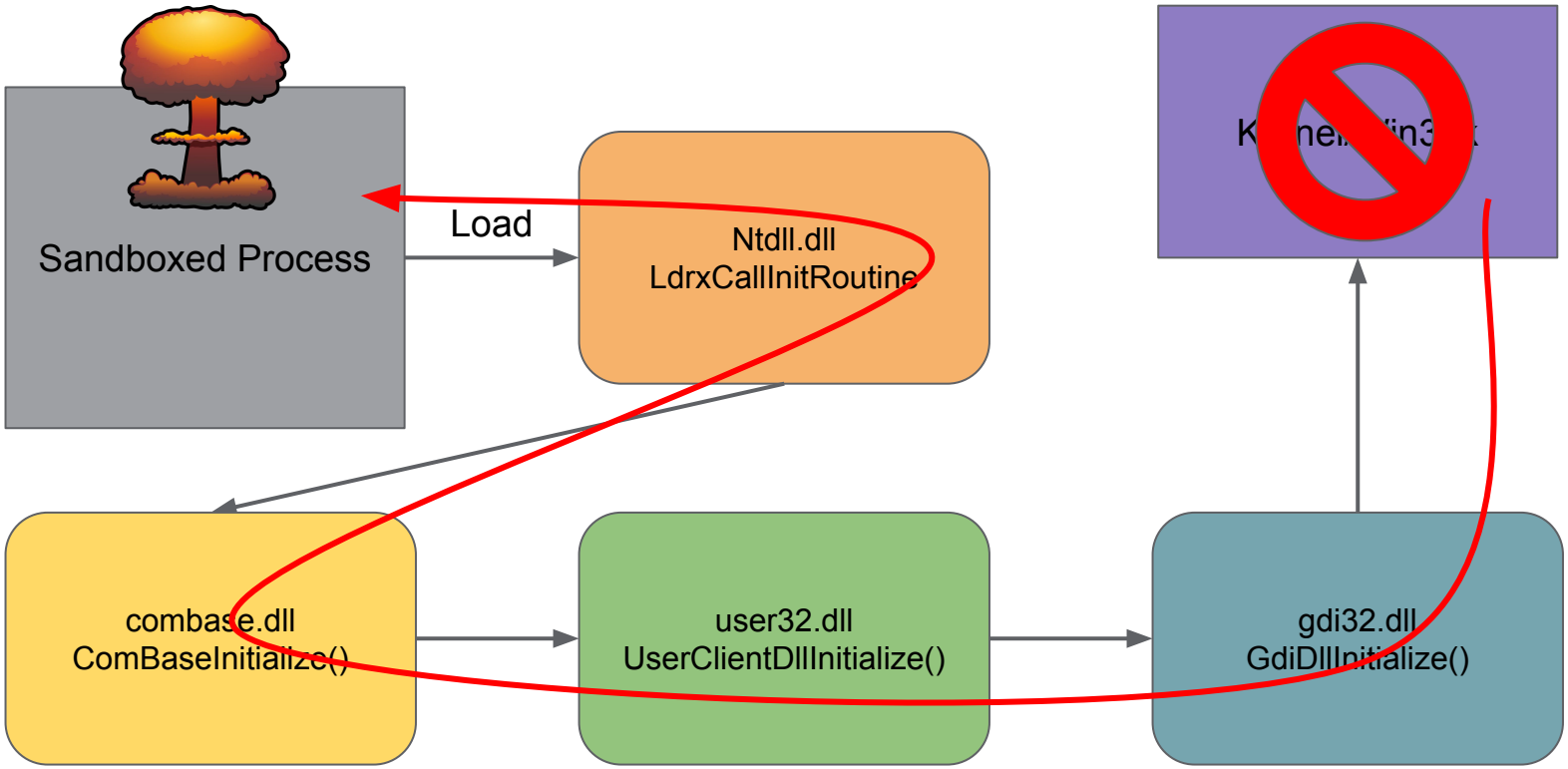


We are too late for this thread, but it' ll still enable the mitigation for the process

Set at Process Creation



Set at Process Creation

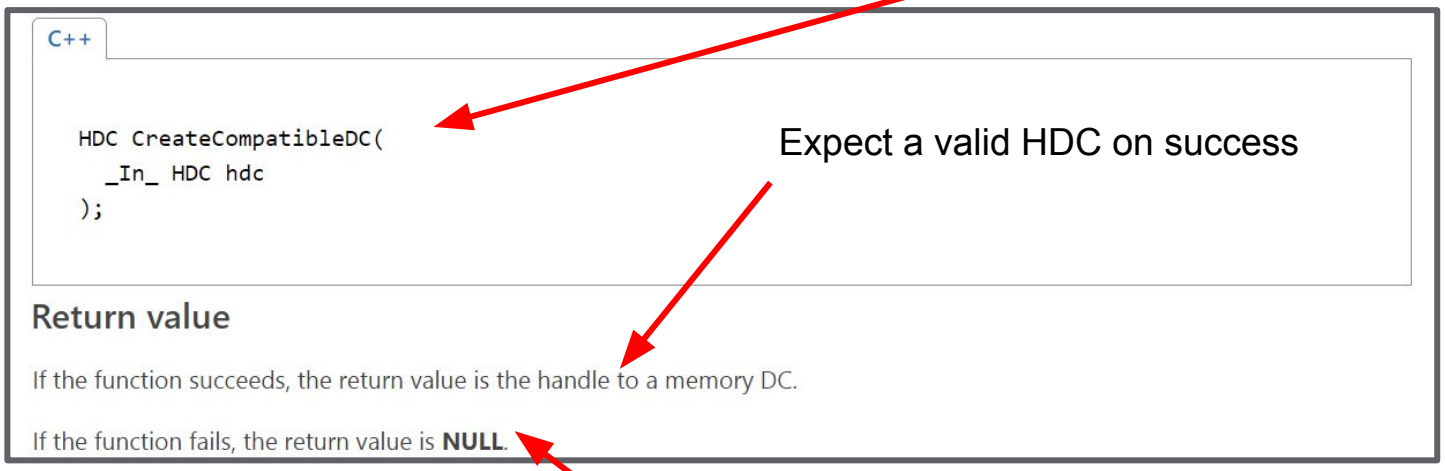


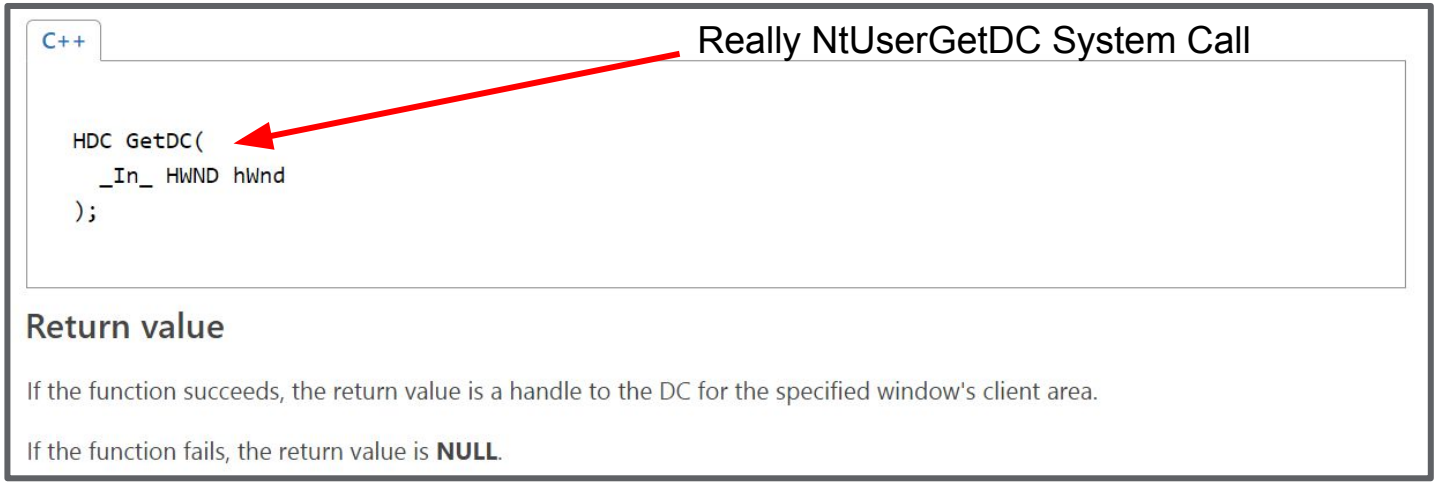
How Does Chrome Do It?

- Chrome hooks the NtMapViewOfSection system call in NTDLL.
- Patches GdiDllInitialize before anything has a chance to call it.

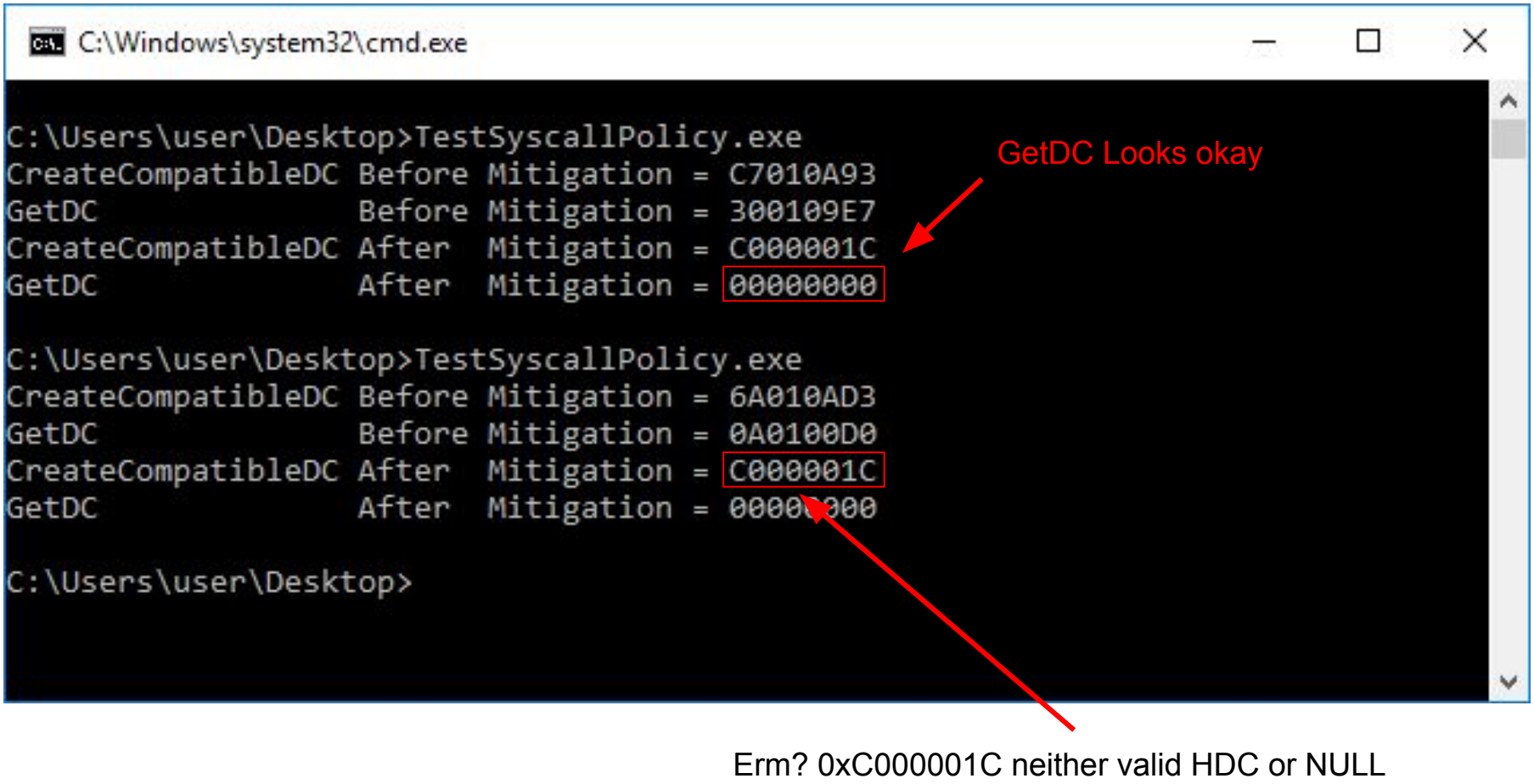
```
BOOL WINAPI TargetGdiDllInitialize (
    GdiDllInitializeFunction orig_gdi_dll_initialize ,
    HANDLE dll,
    DWORD reason) {
  return TRUE;
}

HGDIOBJ WINAPI TargetGetStockObject (
    GetStockObjectFunction orig_get_stock_object ,
    int object) {
  return reinterpret_cast<HGDIOBJ>(NULL);
}

ATOM WINAPI TargetRegisterClassW (
    RegisterClassWFunction orig_register_class_function ,
    const WNDCLASS* wnd_class) {
  return TRUE;
}
```

Syscall Return Values





Let's Test


C:\Windows\system32\cmd.exe
C:\Users\user\Desktop>TestSyscallPolicy.exe
CreateCompatibleDC Before Mitigation = C7010A93
GetDC Before Mitigation = 300109E7
CreateCompatibleDC After Mitigation = C000001C
GetDC After Mitigation = 00000000
C:\Users\user\Desktop>TestSyscallPolicy.exe
CreateCompatibleDC Before Mitigation = 6A010AD3
GetDC Before Mitigation = 0A0100D0
CreateCompatibleDC After Mitigation = C000001C
GetDC After Mitigation = 0000
C:\Users\user\Desktop>
GetDC Looks okay
Erm? 0xC000001C neither valid HDC or NULL

Handling Return Code

Return code map at end of shadow table

```
INT32 GetWin32kResult(DWORD SyscallNo) {
  if (SyscallNo & 0xFFF < ShadowTableLength) {
    INT8* ReturnCodes =
       (INT8*)&ShadowTable[ShadowTableLength];
    INT32 Result = ReturnCodes[SyscallNo & 0xFFF];
    if (Result <= 0)
      return Result;
  }
  return STATUS_INVALID_SYSTEM_SERVICE;
}
```

Pass through value is <= 0

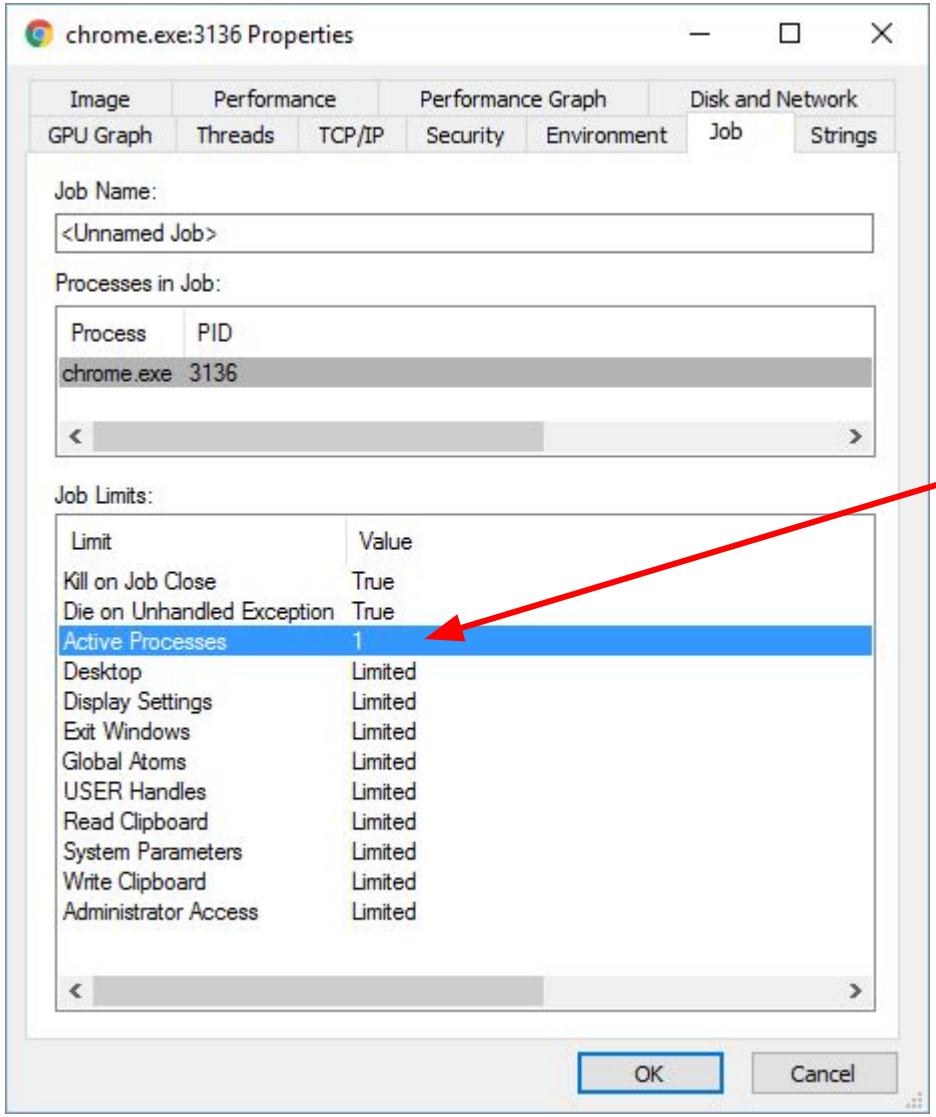
Just happens to equal 0xC000001C, returned if Result > 0

Process Mitigations Inheritance

- No policies can be disabled once set in-process.
- However only a small subset of mitigations are inherited
- See *PspApplyMitigationOptions.*
- we need to block new process creation.

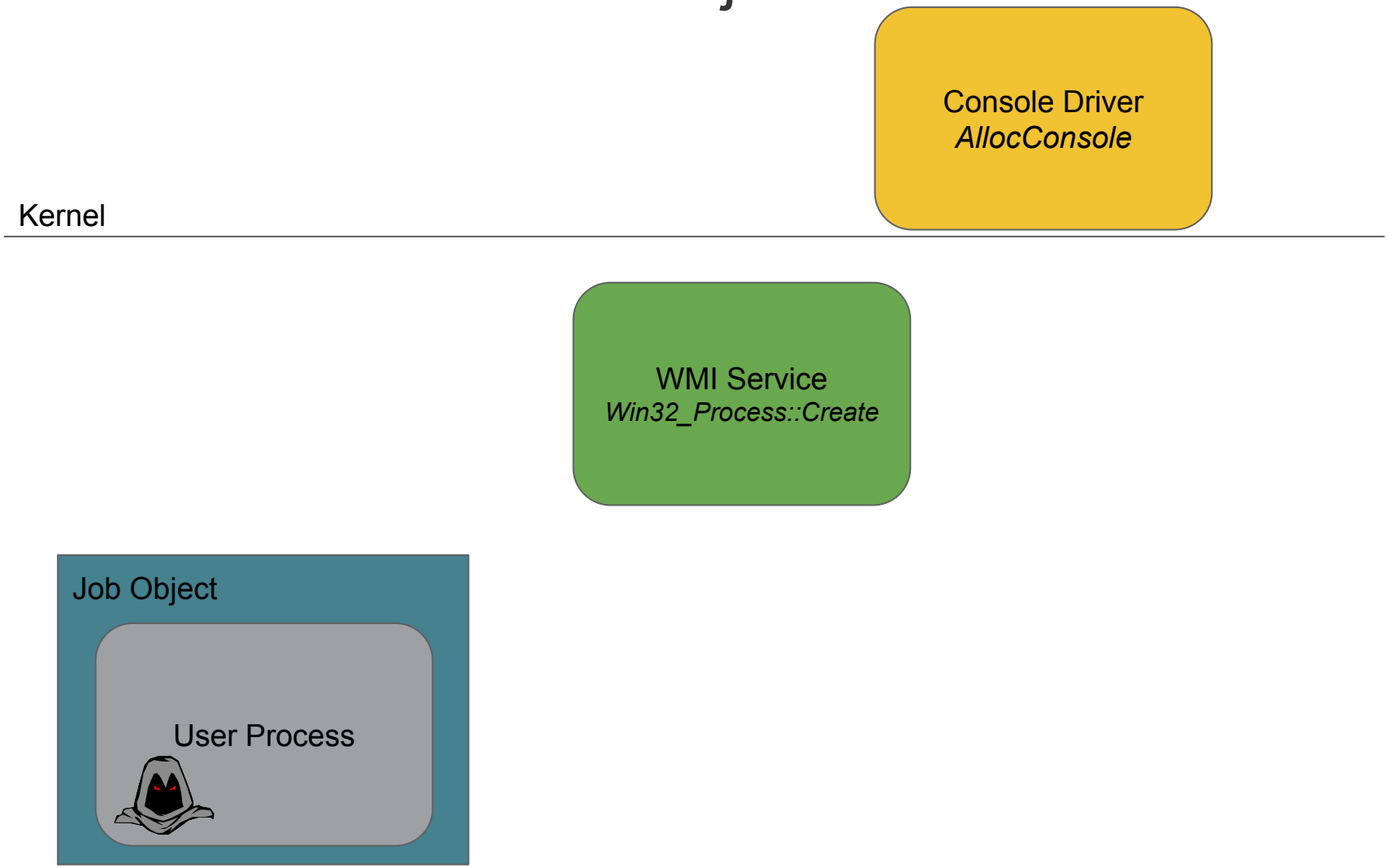
Policy	*Inherited*
Dynamic Code	No
System Call Disable	Yes
Signature	No
Font Disable	No
Image Load	Yes

Job Objects to the Rescue

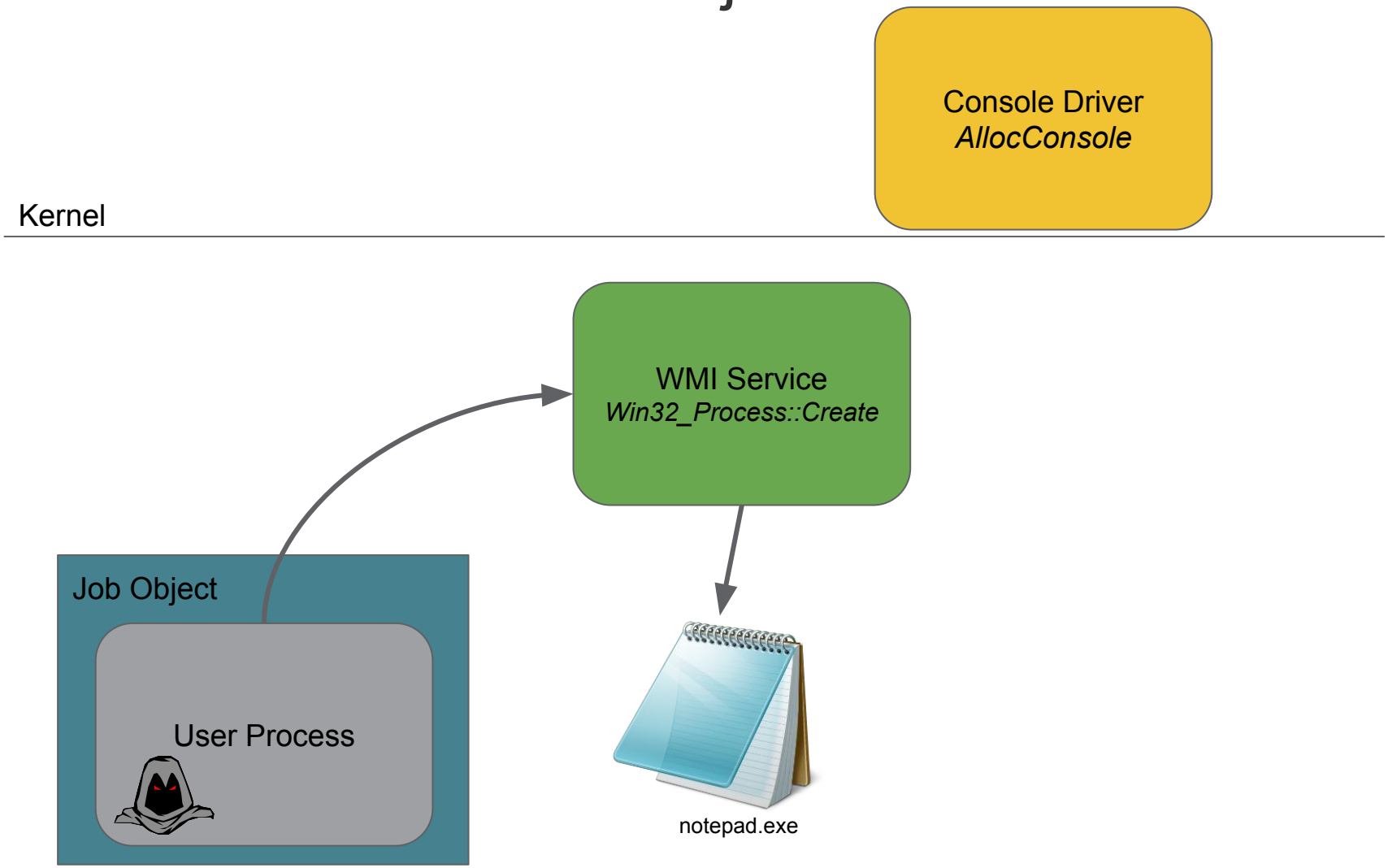


1 Active Process
No Breakout Allowed
Can't create new process

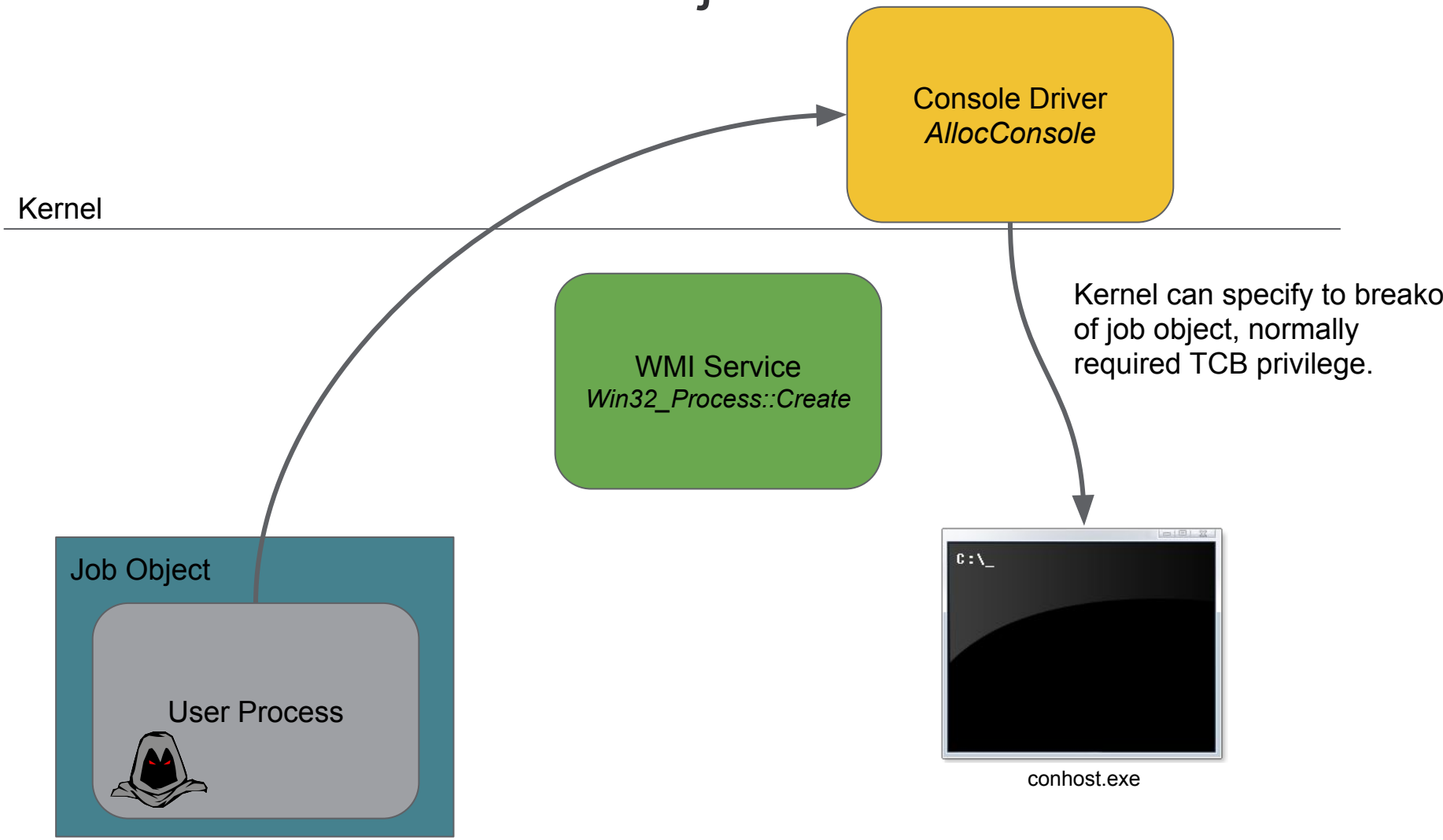
The Trouble with Job Objects



The Trouble with Job Objects

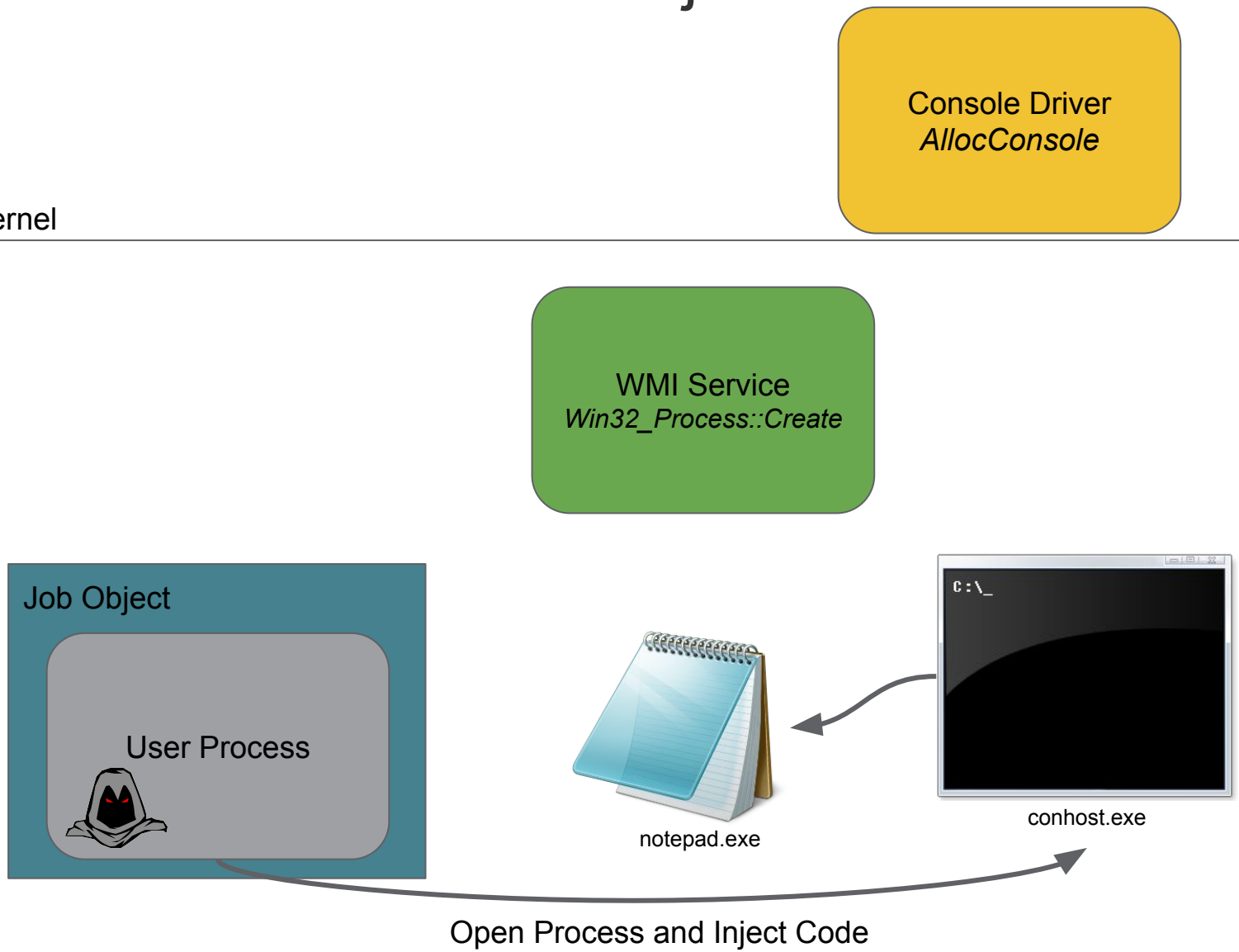


The Trouble with Job Objects



https://bugs.chromium.org/p/project-zero/issues/detail?id=213

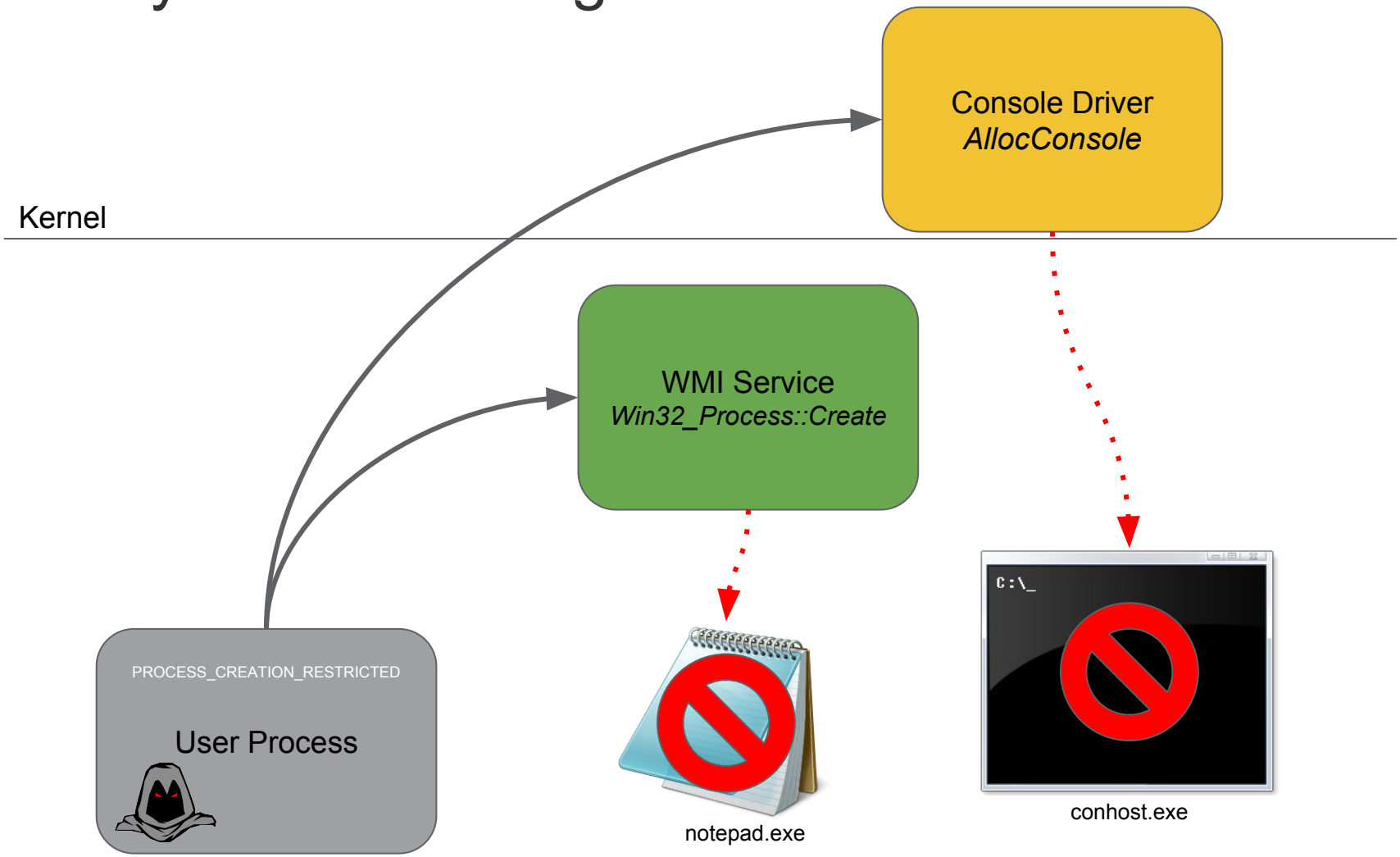
The Trouble with Job Objects



New Process Attribute for Win10 TH2

```
const int ProcThreadAttributeChildProcessPolicy   = 14;
//
// Define Attribute to disable creation of child process
//

#define PROCESS_CREATION_CHILD_PROCESS_RESTRICTED   0x01
#define PROCESS_CREATION_CHILD_PROCESS_OVERRIDE     0x02

#define PROC_THREAD_ATTRIBUTE_CHILD_PROCESS_POLICY \
 ProcThreadAttributeValue(
        ProcThreadAttributeChildProcessPolicy,
        FALSE, TRUE, FALSE)
```

Pretty Effective Mitigation



Inside SeSubProcessToken

Mitigation is on Token not Process

```
DWORD ChildProcessPolicyFlag = // From process attribute.
BOOLEAN ChildProcessAllowed = TokenObject->TokenFlags &
                                  CHILD_PROCESS_RESTRICTED;
if (!ChildProcessAllowed) {
  if (!(ChildProcessPolicyFlag &
              PROCESS_CREATION_CHILD_PROCESS_OVERRIDE)
      || !SeSinglePrivilegeCheck(SeTcbPrivilege))
     return STATUS_ACCESS_DENIED;
}

SepDuplicateToken(TokenObject, ..., &NewTokenObject);

if (ChildProcessPolicyFlag &
            PROCESS_CREATION_CHILD_PROCESS_RESTRICTED)
   NewTokenObject->TokenFlags |= CHILD_PROCESS_RESTRICTED;
```

Low Box Tokens (AppContainers)

- Many specific checks in the kernel for lowbox tokens
  - No NULL DACL access
  - Limited access to ATOM tables
- Capabilities which prevent access to things like network stack
- No read-up by default
- COM marshaling mitigations
  - Marshaled objects from AppContainers are untrusted
- Driver Traversal Check
  - Undocumented FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL device characteristic

Implicit Mitigations

ExIsRestrictedCaller

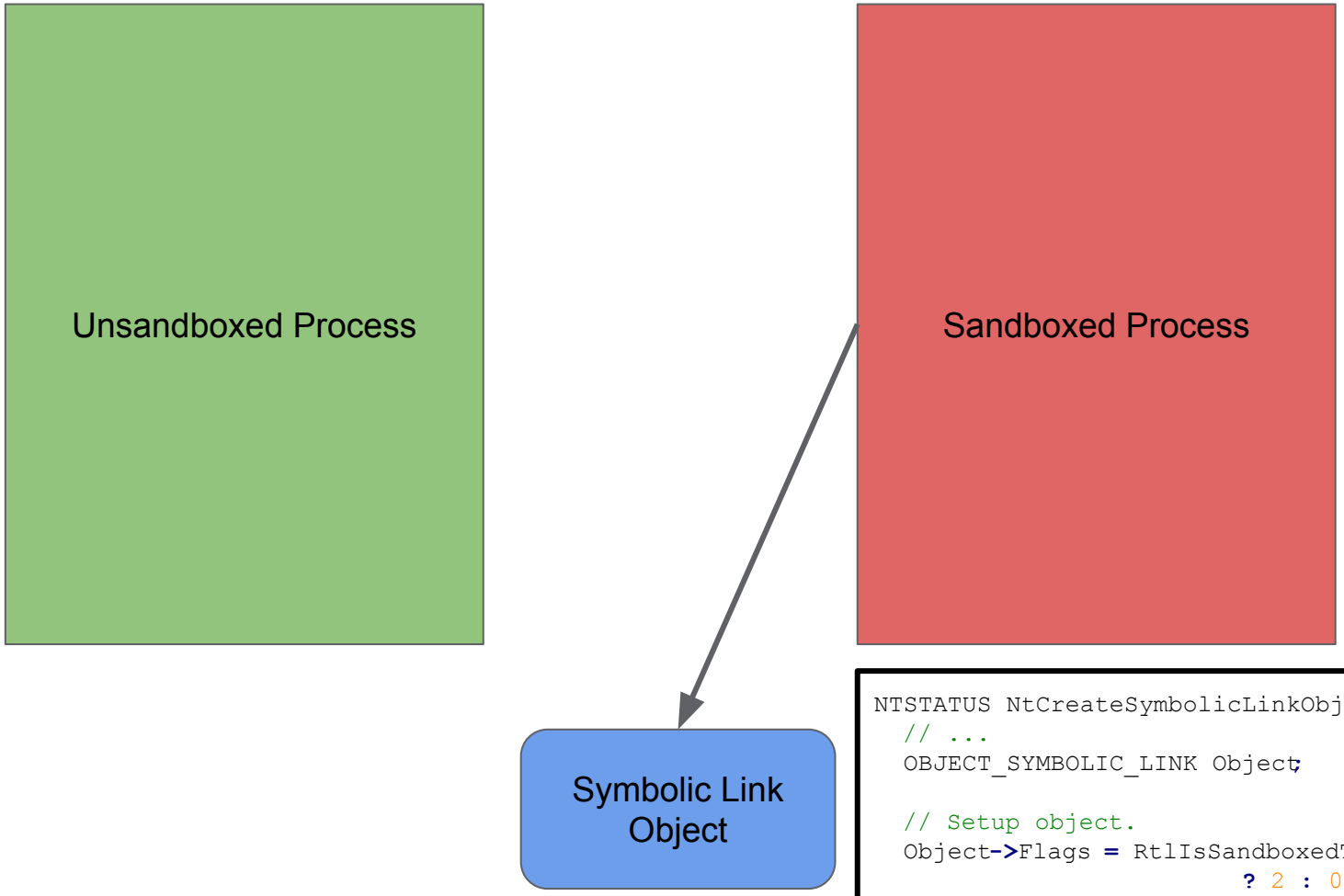
- Kernel function introduced in Windows 8.1
- Indicates whether the caller is “restricted”, what this actually means in practice is whether the caller has an IL < Medium or is a Low Box Token
- Only used to filter out sensitive kernel addresses to limit KASLR leaks
  - Blocks profiling kernel addresses
  - Limits access to handle list which contains object addresses
  - Limits access to object type information
  - Limits thread and process information which leaks kernel addresses
- More information on Alex Ionescu’s blog:
  - http://www.alex-ionescu.com/?p=82

RtlIsSandboxedToken

- Very similar function to ExIsRestrictedCaller, but passes an token
- Introduced in Windows 10 but backported to Windows 7

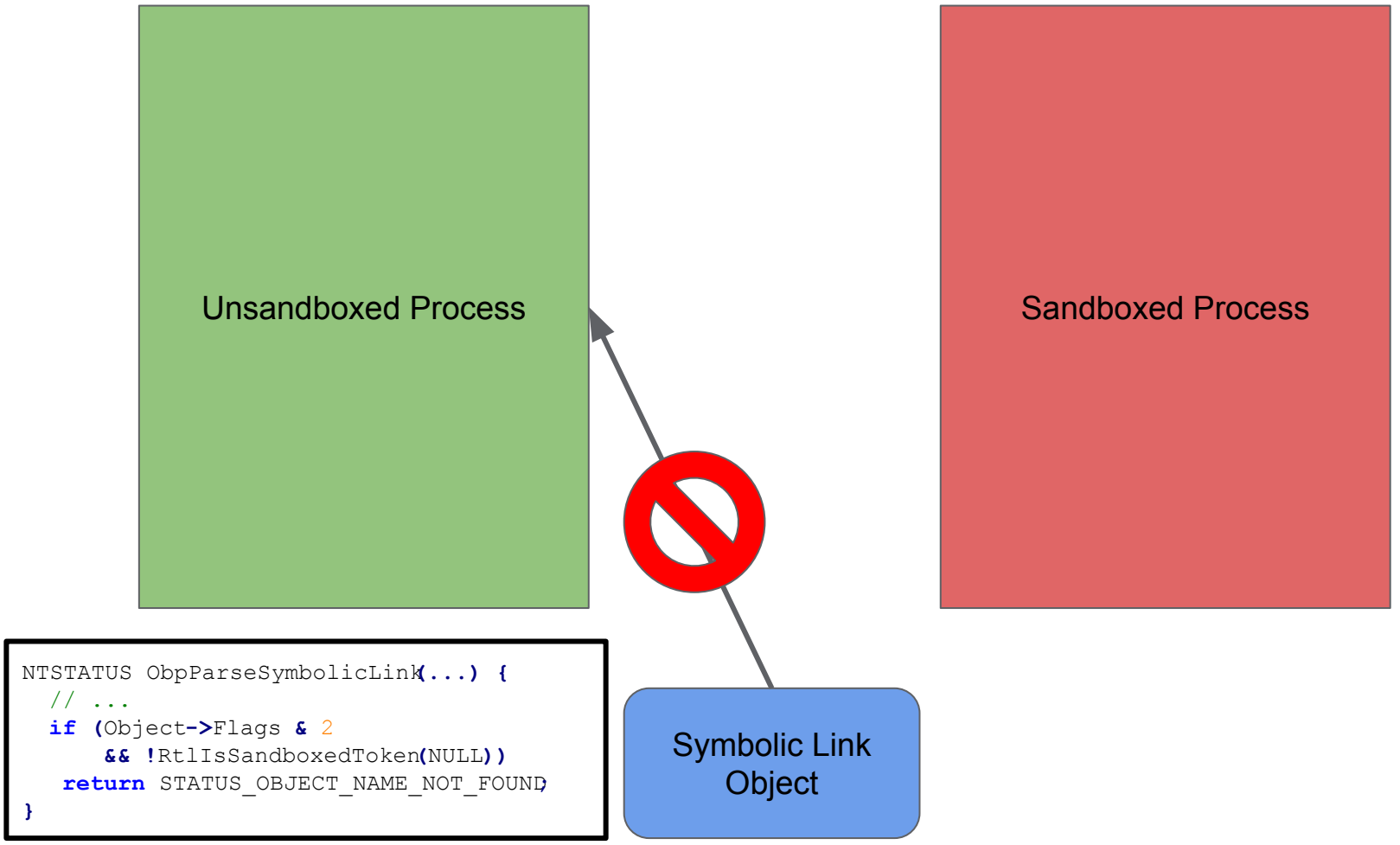
```
BOOLEAN RtlIsSandboxedToken(PSECURITY_SUBJECT_CONTEXT
                            SubjectSecurityContext) {
  SECURITY_SUBJECT_CONTEXT SecurityContext;

  if (!SubjectSecurityContext) {
    SeCaptureSubjectContext(&SecurityContext);
    SubjectSecurityContext = &SecurityContext;
  }
  return !SeAccessCheck(
         SeMediumDaclSd,
         SubjectSecurityContext,
         0x20000u);
}
```

Blocking Object Manager Symbolic Links




```
NTSTATUS NtCreateSymbolicLinkObject(...) {
  // ...
  OBJECT_SYMBOLIC_LINK Object;

  // Setup object.
  Object->Flags = RtlIsSandboxedToken(NULL)
                              ? 2 : 0;
}
```

Blocking Object Manager Symbolic Links




```
NTSTATUS ObpParseSymbolicLink(...) {
  // ...
  if (Object->Flags & 2
      && !RtlIsSandboxedToken(NULL))
   return STATUS_OBJECT_NAME_NOT_FOUND;
}
```

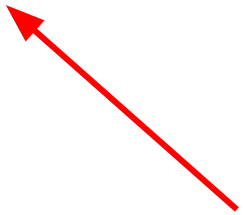
Blocking Registry Key Symbolic Links

```
NTSTATUS CmpCheckCreateAccess(...) {
  BOOLEAN AccessGranted = SeAccessCheck(...);
  if (AccessGranted &&
      CreateOptions & REG_OPTION_CREATE_LINK &&
      RtlIsSandboxedToken()) {
    return STATUS_ACCESS_DENIED;
  }
}
```

```
NTSTATUS CmSetValueKey(...) {
  if(Type == REG_LINK &&
     RtlEqualUnicodeString(&CmSymbolicLinkValueName,
                           ValueName, TRUE) &&
     RtlIsSandboxedToken())
     return STATUS_ACCESS_DENIED;
}
```

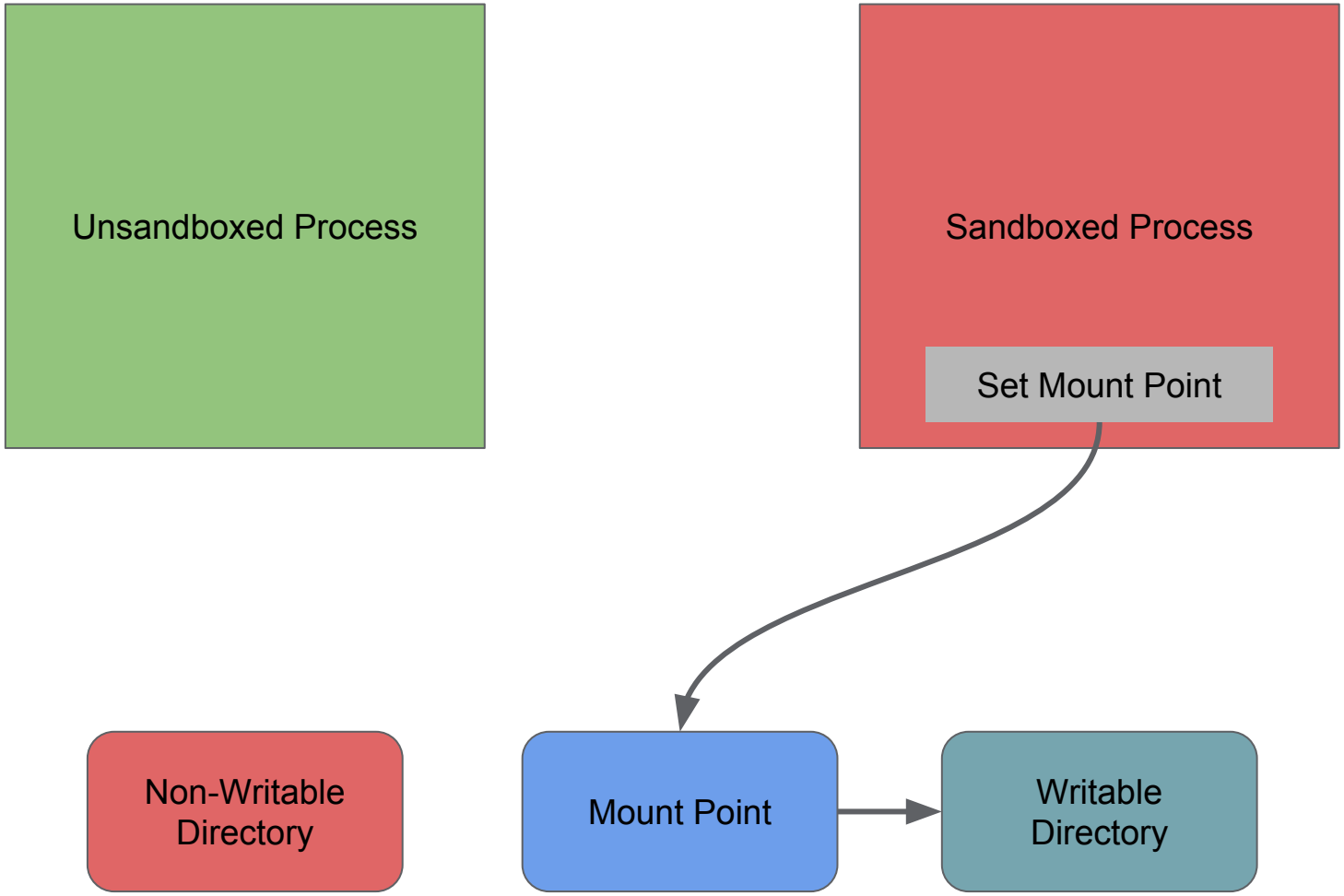
Blocking NTFS Mount Points

```
NTSTATUS IopXxxControlFile(...) {
  if (ControlCode == FSCTL_SET_REPARSE_POINT &&
      RtlIsSandboxedToken()) {
    status = FsRtlValidateReparsePointBuffer(buffer);
    if (status < 0)
      return status;
    if (buffer.ReparseTag == IO_REPARSE_TAG_MOUNT_POINT) {
      InitializeObjectAttributes(&ObjAttr,
                  reparse_buffer.PathBuffer, ...);
      status = ZwOpenFile(&FileHandle, FILE_GENERIC_WRITE,
            &ObjAttr, ..., FILE_DIRECTORY_FILE);
      if (status < 0)
        return status;
       // Continue.
    }
  }
}
```

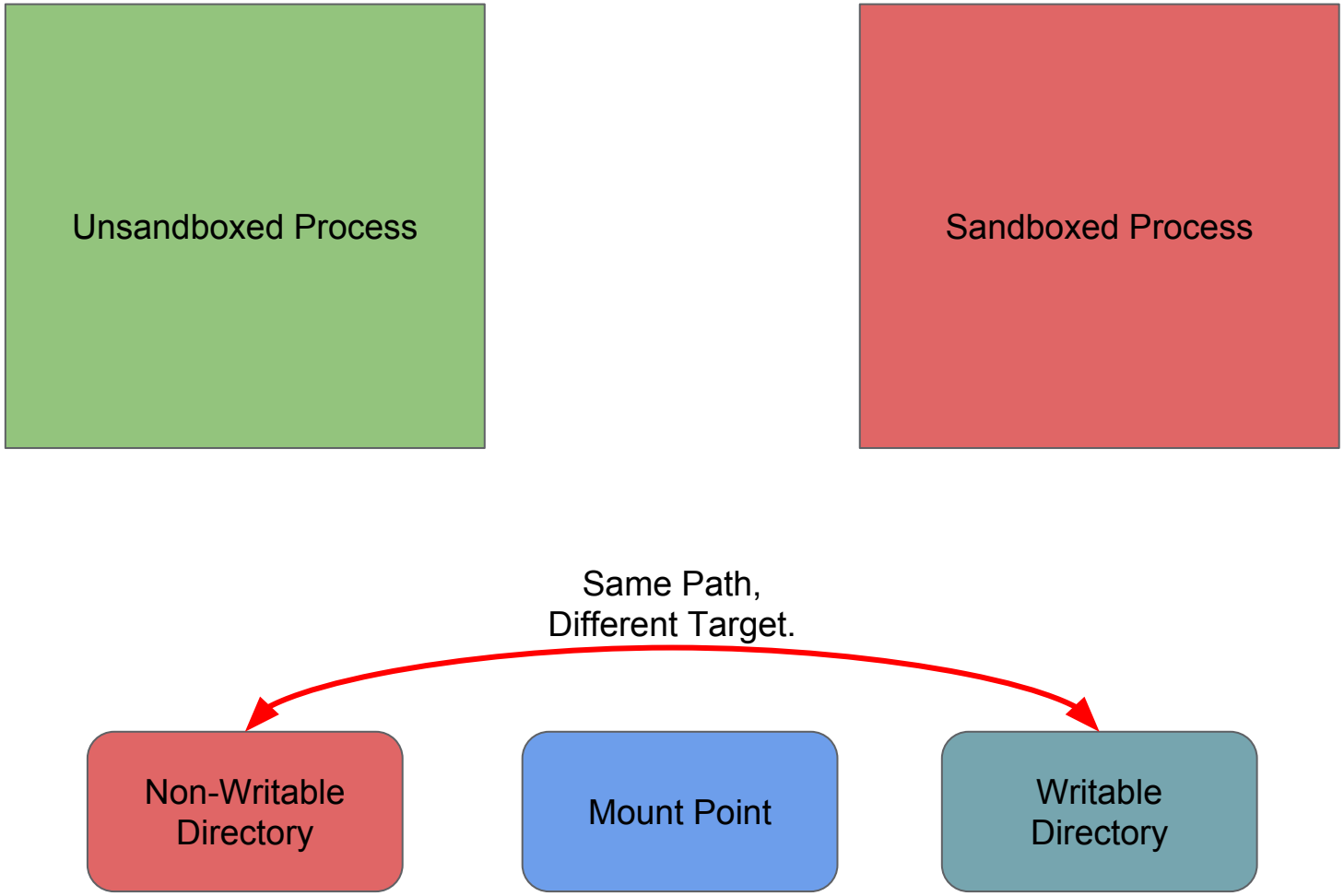


Checks target is a directory and writable

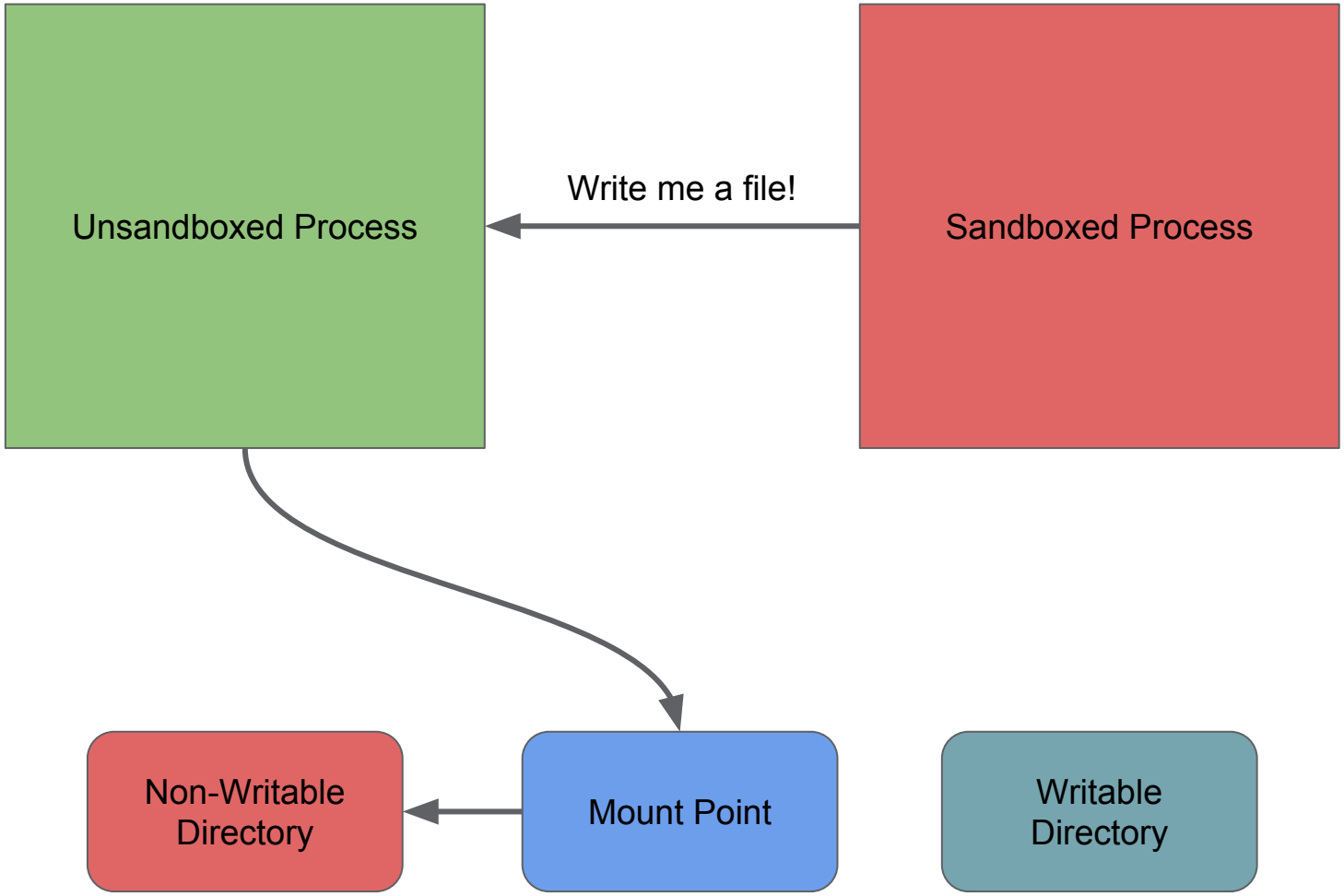
TOCTOU Setting Mount Point



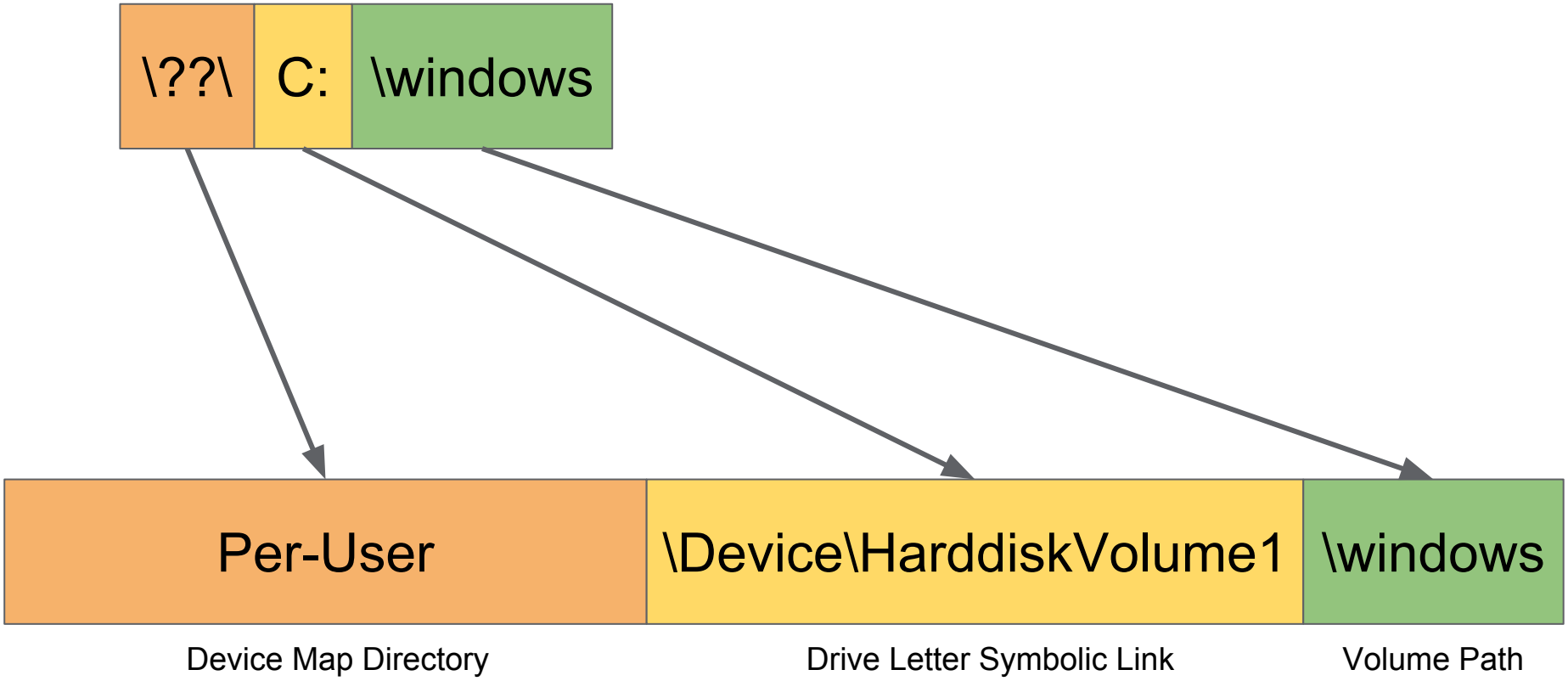
TOCTOU Setting Mount Point



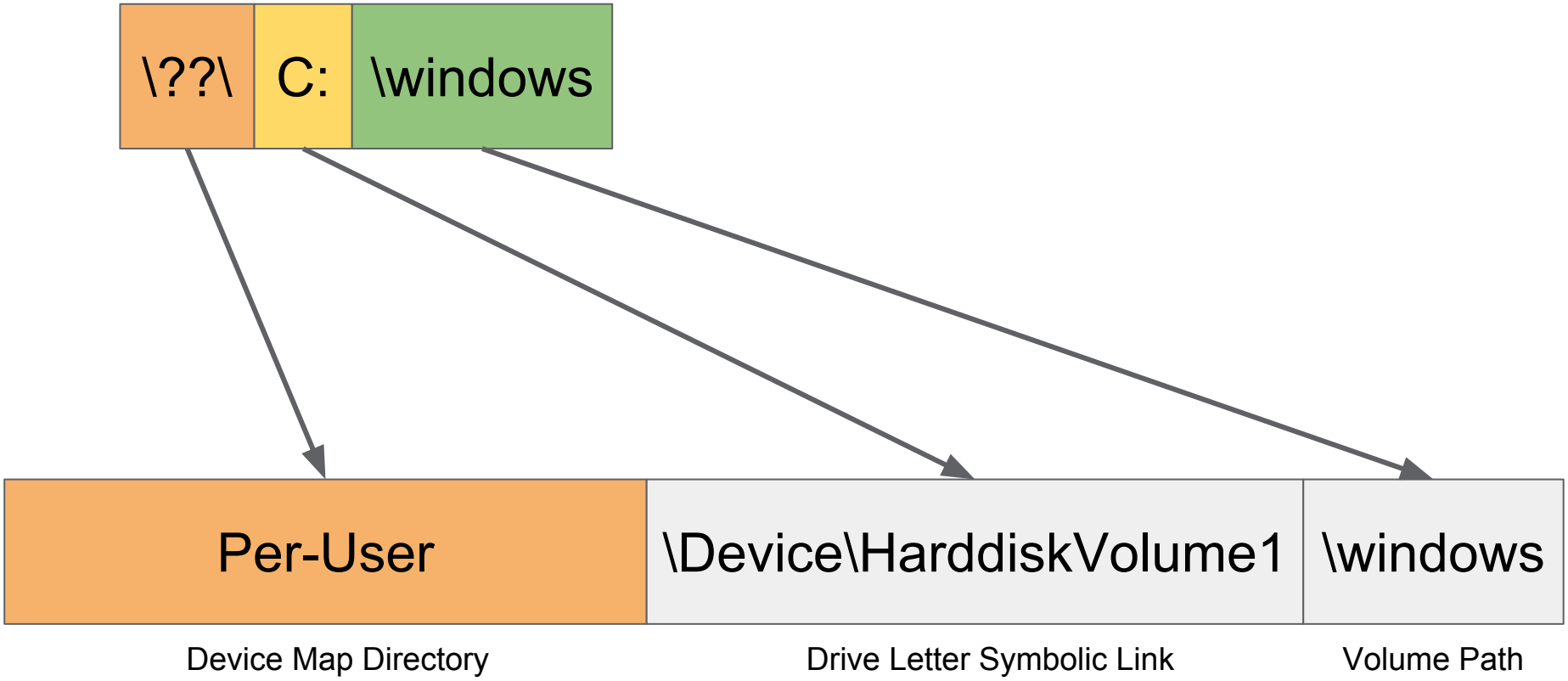
TOCTOU Setting Mount Point



Blocking NTFS Mount Points



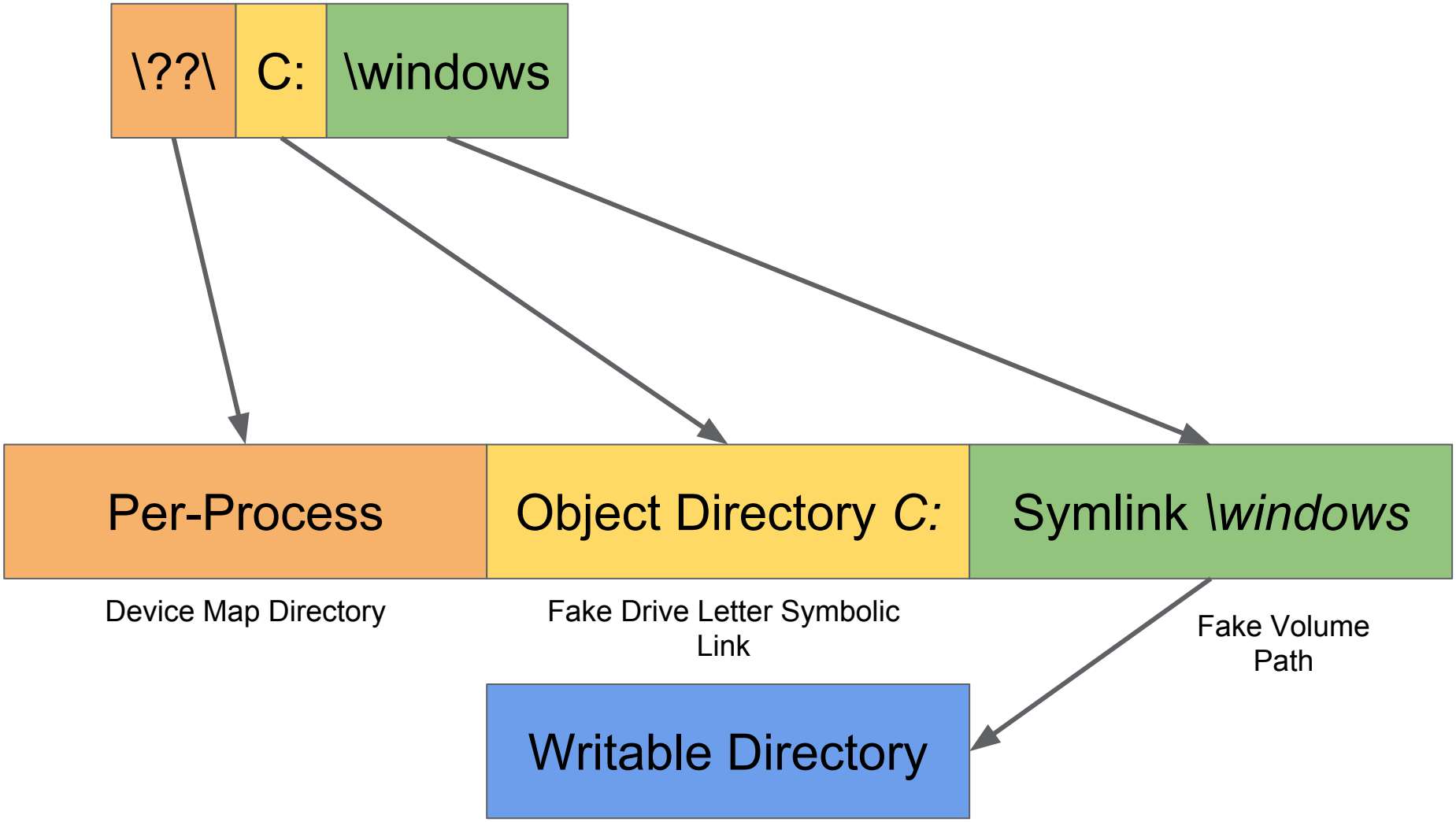
Blocking NTFS Mount Points



Per-Process Device Map

```
const int ProcessDeviceMap = 23;

struct PROCESS_DEVICEMAP_INFORMATION {
    HANDLE DirectoryHandle;
};

bool SetProcessDeviceMap(HANDLE hDir) {
    PROCESS_DEVICEMAP_INFORMATION DeviceMap = {hDir};
    NTSTATUS status = NtSetInformationProcess(
                          GetCurrentProcess(),
                          ProcessDeviceMap,
                          &DeviceMap,
                          sizeof(DeviceMap));
    return status == 0;
}
```

Blocking NTFS Mount Points

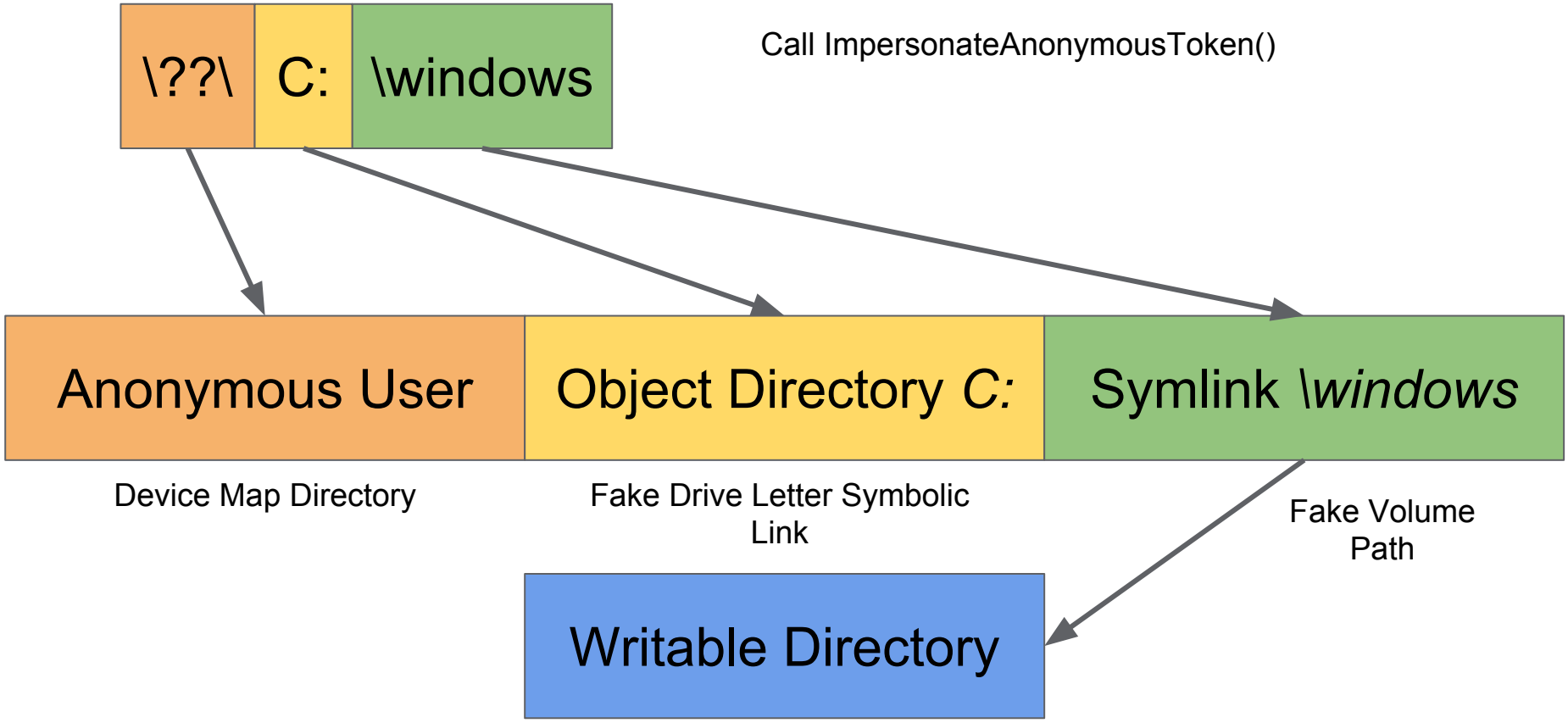


Blocking Process Device Map

```
NTSTATUS NtSetInformationProcess(...) {
    // ...

    case ProcessDeviceMap:
      HANDLE hDir = *(HANDLE*)Data;
      if (RtlIsSandboxedToken())
        return STATUS_ACCESS_DENIED;
      return ObSetDeviceMap(ProcessObject, hDir);

    // ...
}
```

https://bugs.chromium.org/p/project-zero/issues/detail?id=486

Bypass Using Anonymous Token



https://bugs.chromium.org/p/project-zero/issues/detail?id=573
https://bugs.chromium.org/p/project-zero/issues/detail?id=589

NTFS Hard Links

CreateHardLink function

Establishes a hard link between an existing file and a new file. This function is only supported on the NTFS file system, and only for files, not directories.

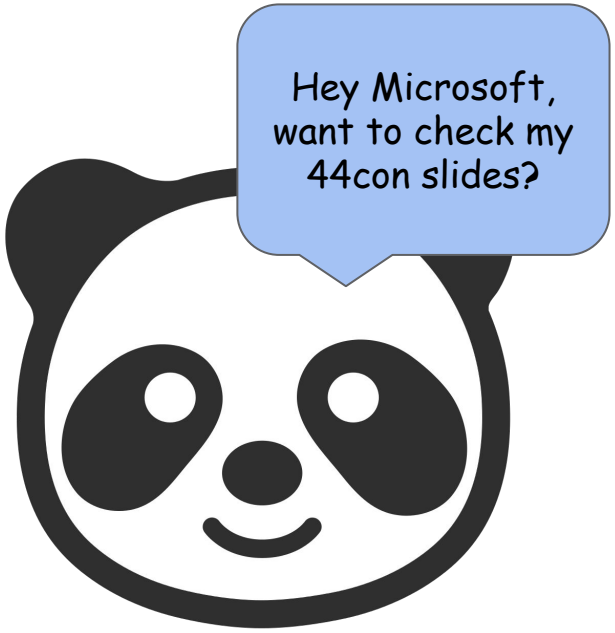
To perform this operation as a transacted operation, use the CreateHardLinkTransacted function.

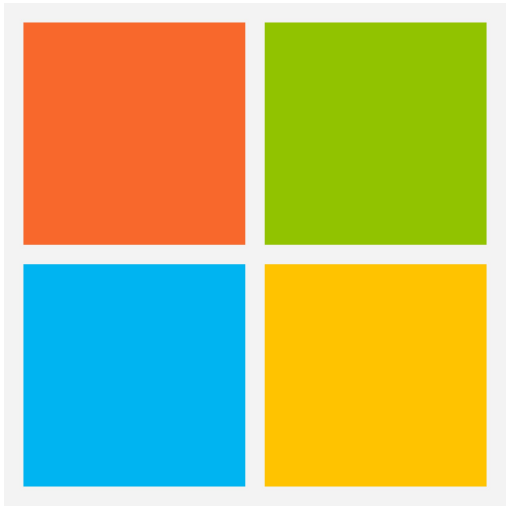
Syntax

C++

```
BOOL WINAPI CreateHardLink(
  _In_       LPCTSTR               lpFileName,
  _In_       LPCTSTR               lpExistingFileName,
  _Reserved_ LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Never Give a Vendor a Preview of your Slides





Never Give a Vendor a Preview of your Slides

Never Give a Vendor a Preview of your Slides

Blocking NTFS Hardlinks

```
NTSTATUS NtSetInformationFile(...) {
  case FileLinkInformation:
    ACCESS_MASK RequiredAccess = 0;
    if(RtlIsSandboxedToken()) {
      RequiredAccess |= FILE_WRITE_ATTRIBUTES;
    }

    ObReferenceObjectByHandle(FileHandle, RequiredAccess);
}
```

Default, no access requires

In sandbox needs write access

https://bugs.chromium.org/p/project-zero/issues/detail?id=531

The Ultimate Set of Options for Win10

- At least run with a lowbox token, ideally lowbox+restricted
  - Kicks in implicit sandbox mitigations
- Disable dynamic code
- Only allow Microsoft signed images to be loaded
- Disable child process creation
- Win32k Lockdown or at least block custom fonts
- Prevent loading DLLs from untrusted locations

Wrapping Up

Conclusions

- Microsoft adding more and more sandbox related mitigations to the core of Windows. Still a lot more could be done though.
- They’re actively fixing issues with the mitigations, something they’ ve never really done before.
- You can make a fairly secure sandbox, especially with attack surface reduction.

References and Source Code

- Sandbox Analysis Tools
  - https://github.com/google/sandbox-attacksurface-analysis-tools
- Symbolic Link Testing Tools
  - https://github.com/google/symboliclink-testing-tools
- Chrome Windows Sandbox Code
  - https://code.google.com/p/chromium/codesearch#chromium/src/sandbox/win/src/

Questions?