Syringe:
Extraterrestrial Control of Virtual Machines

Martim Carbone, Georgia Tech Ph.D. Candidate

Matt Conover, Technical Director (Software Architect)

Core Research Group, Symantec Research Labs

February 2011

Why Externally Control Virtual Machines?

symantec™

Why we shouldn’t install security software in virtual machines (VMs):

1. Administrative headache

2. Wasted resources (disk space, CPU time, etc.)

3. Security risk

High-Level Problems

symantec.

- Administrative headache
  - Administrator needs to keep all machines updated
  - Need to install separate agents for everything (an anti-virus agent, a software update agent, etc.)
  - Less-than-seamless: if the user gets infected with a virus, it may disable the anti-virus. Then what? Administrator may need to restore the virtual machine to a previous clean state

High-Level Problems

symantec™

- Wasted resources
  - Why does each desktop need an update program when the enterprise desktops are all fairly homogenous?
  - Antivirus scans all files at least once per VM, although each VM mostly has the same files. The agent of each VM is working in isolation.
  - Having the same software installed on each VM wastes disk space
  - Performing the same scans on each VM wastes CPU resources

High-Level Problems

symantec™

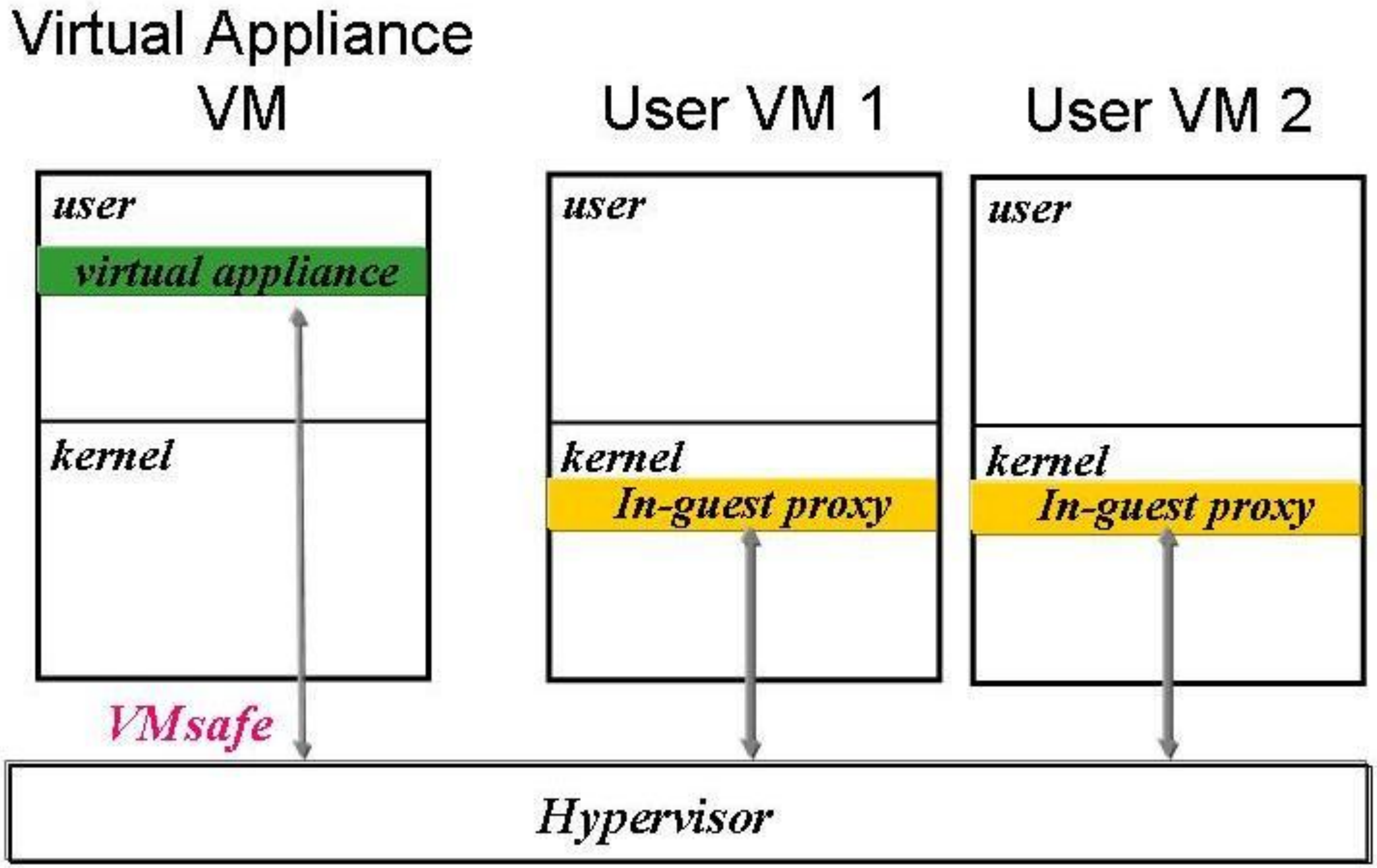
- Security risk
  - The classic problem of security software and threats operating at the same privilege level
  - If the security agent lives on the workstation, it can be disabled by undetected malware.
  - There is no way to reliable way to fix this situation except to boot from a rescue CD

High-Level Solution

- Eliminate the need for software to run within each VM
  - Do shared work from a central location (hypervisor / trusted VM)
    - Example: file scanning
  - However, we still need to be able to do some things inside the guest VM.
    - Example: OS-specific remediation (kill a process)

- Benefits
  - Don't need to maintain software in each VM
  - Perform a task once globally instead of once per VM
  - No software/code in VM (nothing for malware to disable)

Syringe: Architecture

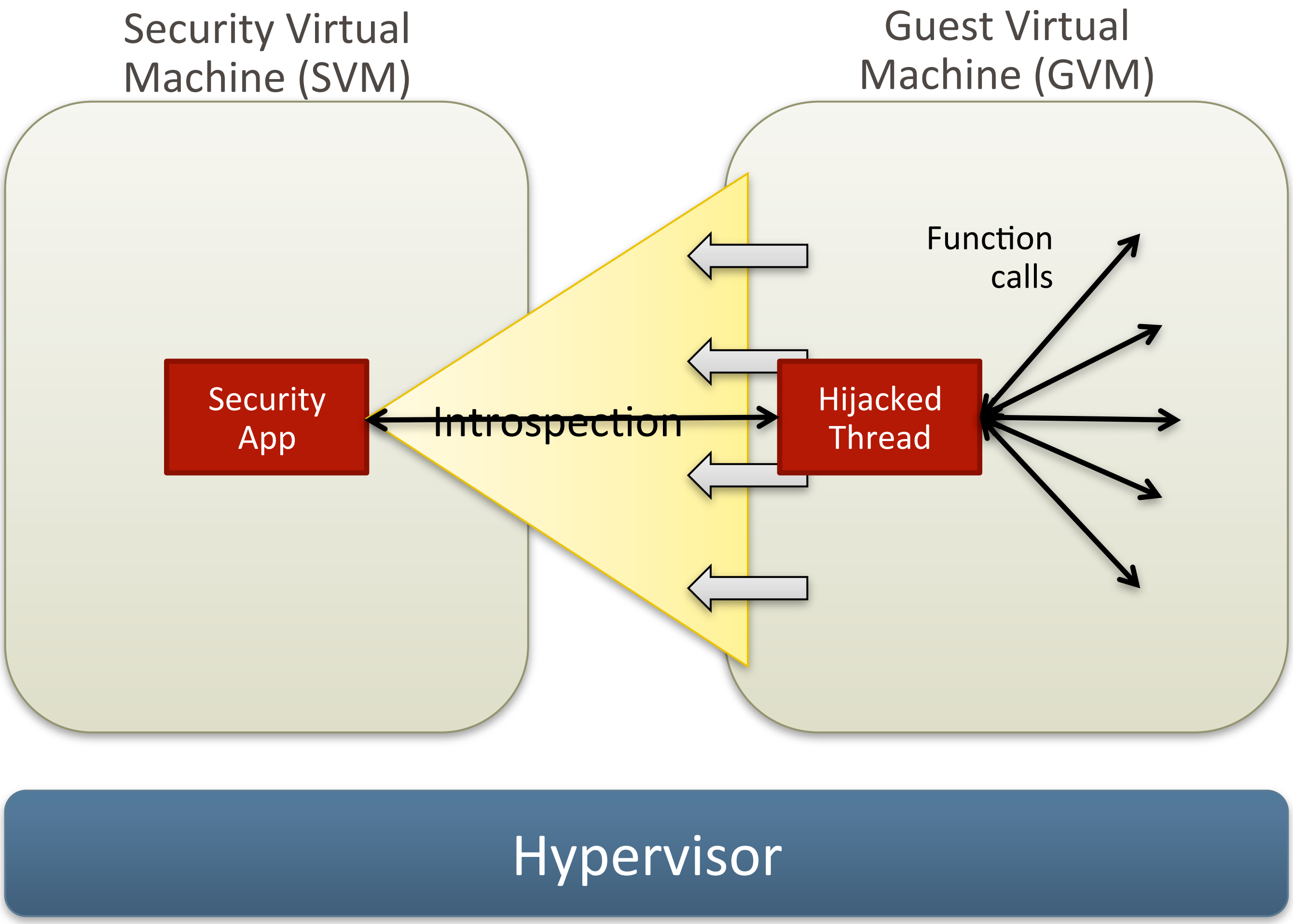


Syringe: Background

- We want to understand what's going on in the virtual machine
  - It's hard to do this properly from outside the virtual machine
  - We see only memory and CPU state (when peering into a virtual machine from the outside)
  - Virtual machine introspection is a hard problem
  - Requires recognizing and understanding the memory.
  - This lack of context is known as the **semantic gap**

- Trivial example: How can we determine the OS version?
  - Inside OS it's easy: call the `RtlGetVersion` API
  - From the outside? Not so easy…

Syringe: Background



Assuming VMware ESX Server + VMsafe for the rest of the talk

Function Call Injection (FCI): Introduction

- What do we want?
  - The ability to invoke arbitrary GVM user/kernel functions from the SVM without having to rely on an in-guest agent (software running in the VM)
  - We want to “borrow” (or hijack) some existing thread to call an API on our behalf, get the result, and then return the hostage thread as if nothing happened
  - Before: A -> B -> C
  - After: A -> B -> Z -> C

Function Call Injection (FCI): Introduction

- Why do we want to do that?

- FCI is like Jedi mind control…
  - Utilizes existing threads, existing drivers and processes to do momentarily our bidding
  - The only way to detect this is to notice a thread deviates from its normal behavior (takes longer, calls different sequence, etc.)
  - For a hostile threat in a GVM to detect and block this, it will essentially be fighting his own shadow (FCI can be performed in any context)

Function Call Injection

- Core technique: ***Function Call Injection***
  1. Use introspection to modify registers (EIP + ESP) and memory (stack)
  2. When GVM resumes execution, it does so as if a function call had just been executed.

  Result: Inter-VM RPC (Remote Procedure Call across virtual machines)

  – Terminology:

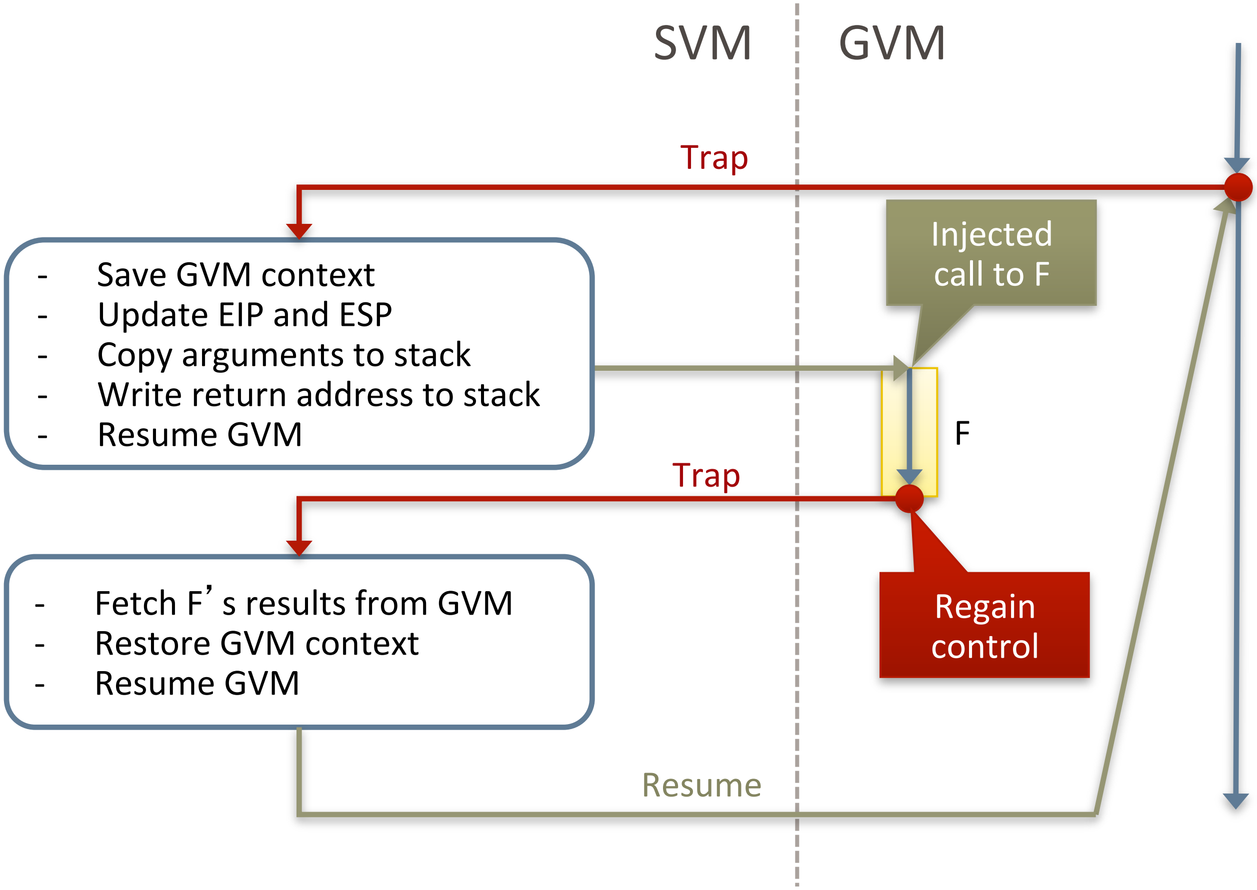
  ESP = Stack Pointer (Stack used by functions to track state)

  EIP = Instruction Pointer (Points to the current executing instruction)

  API = Application Programming Interface (API call / function call)

Function Call Injection

Injecting a call to function F…



- Save GVM context
- Update EIP and ESP
- Copy arguments to stack
- Write return address to stack
- Resume GVM

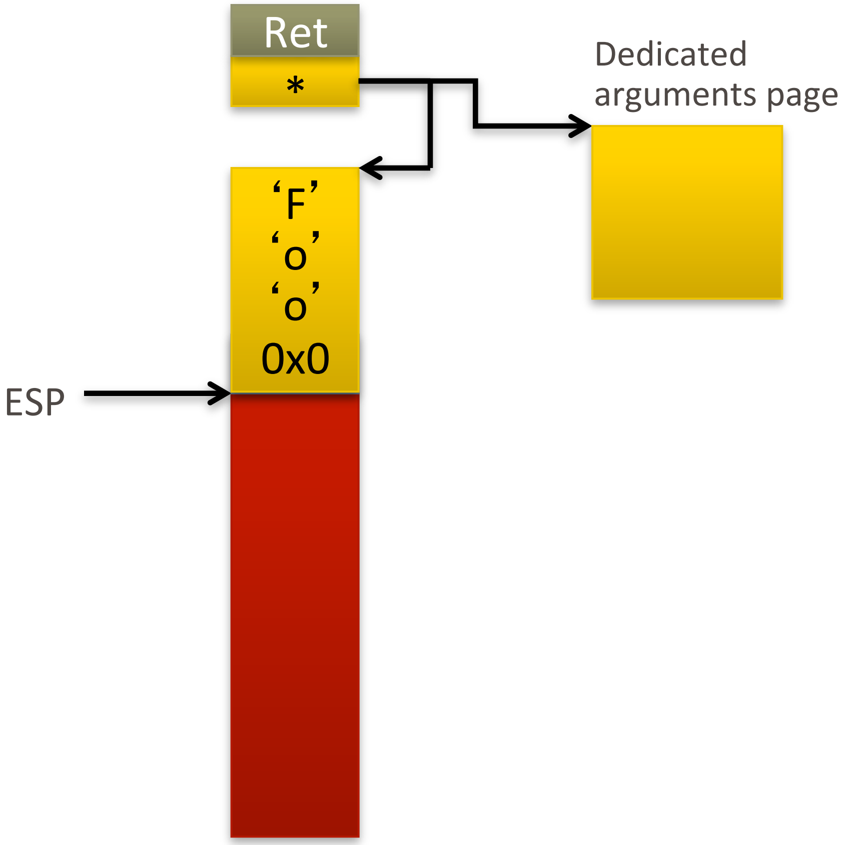
- Fetch F’s results from GVM
- Restore GVM context
- Resume GVM

FCI: Argument Passing on the Stack

SVM

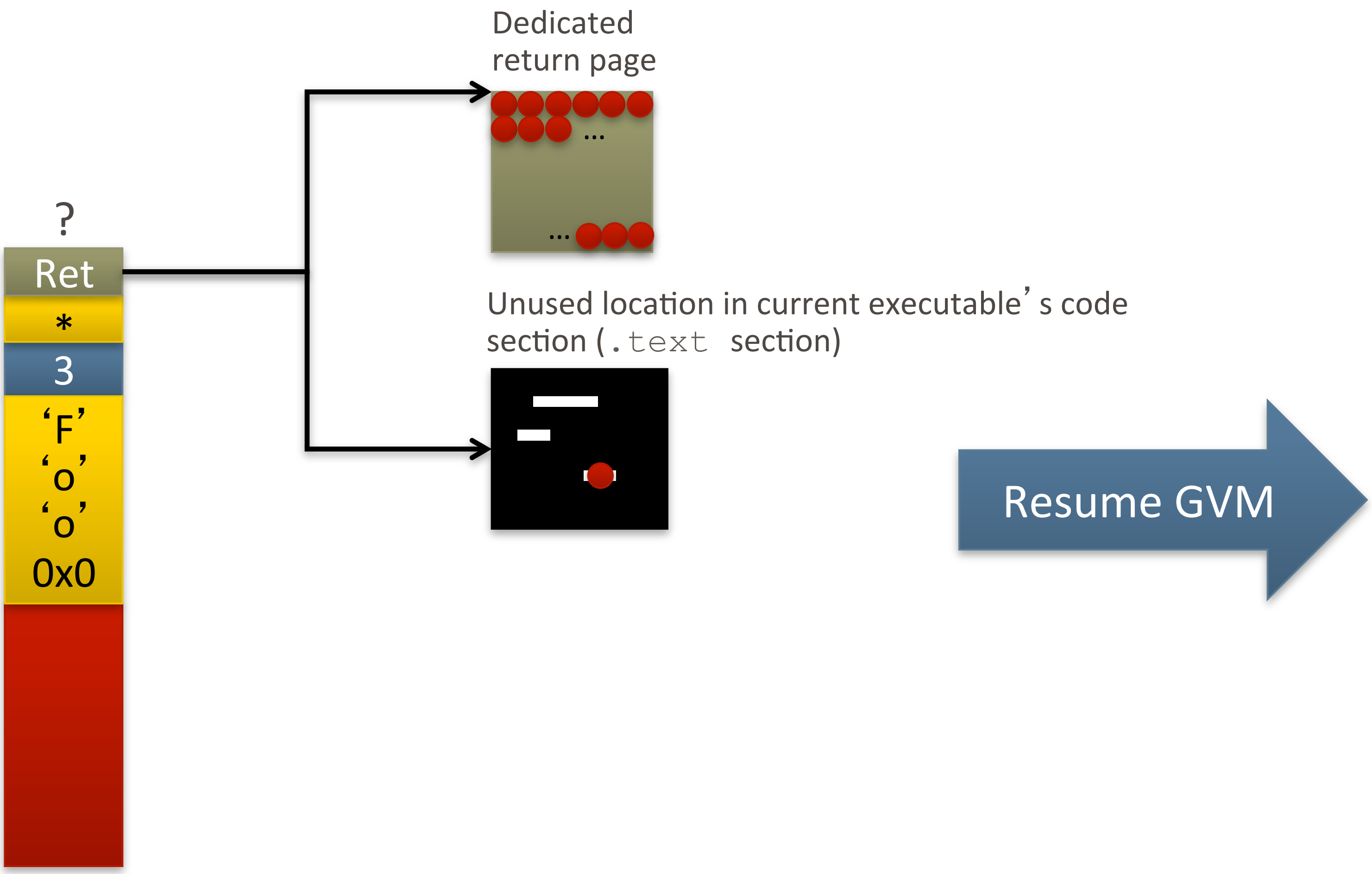
```
F(char *str, int n);

char *s = “Foo”;
F(s, 3);
```

GVM



Current GVM stack

FCI: Returning Control

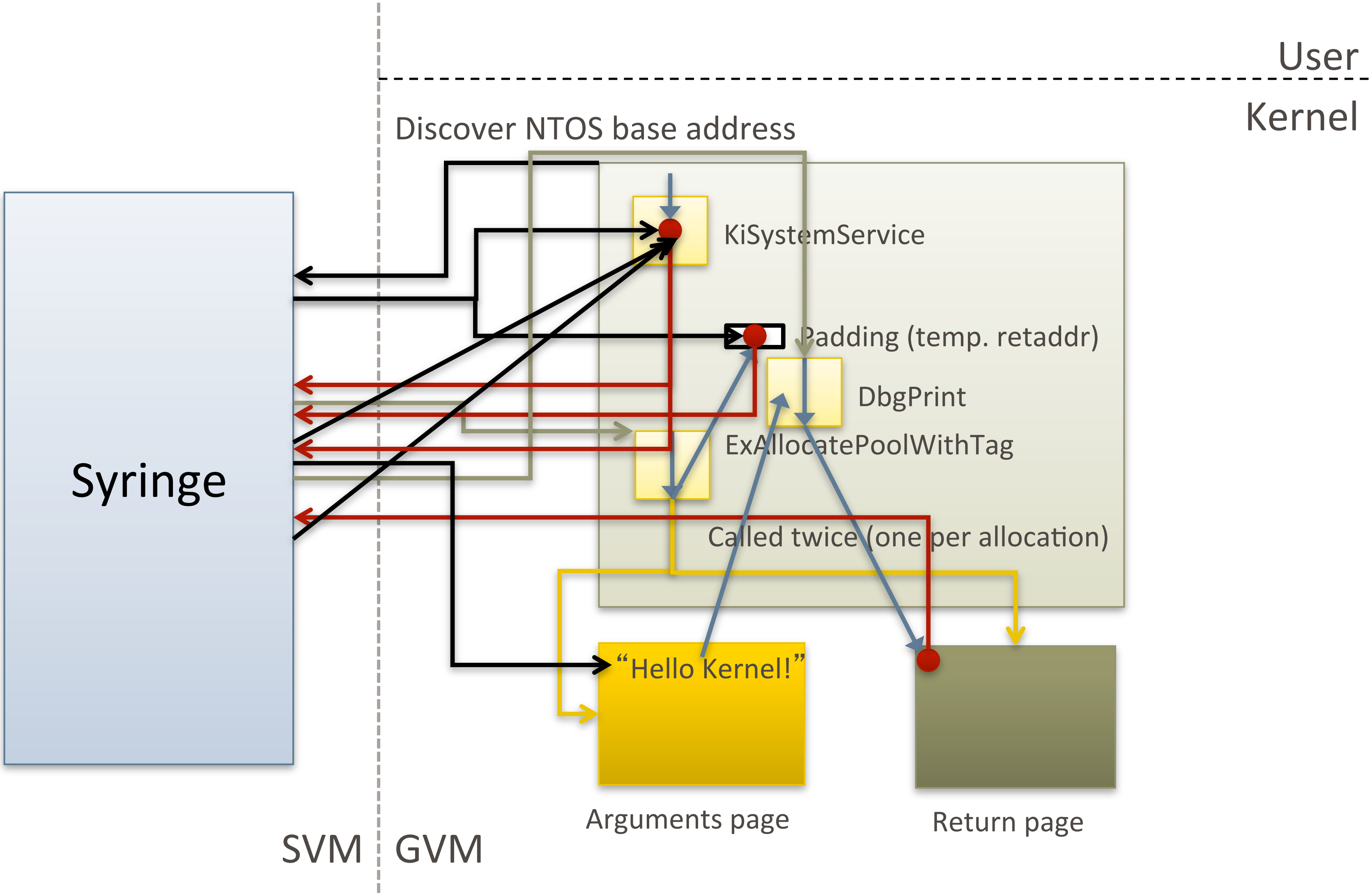


Injection Scope

- Types of in-guest software agents
  - Kernel-mode agent (kernel driver)
    - Invokes public kernel functions (e.g, Windows WDK API)
  - User-mode agent (process)
    - Invokes library API functions (e.g, DLL-based Windows API)
- Our goal is to be able to impersonate both agent types using FCI
  - Kernel-mode FCI
  - User-mode FCI

- Injection point requirements
  - Sufficiently high execution frequency
  - Safe, consistent state (e.g., no locks held that could cause deadlocks)

Kernel-Mode FCI: The Full Process



Kernel-Mode FCI: Example APIs

Ntos!RtlGetVersion() → Retrieve GVM’ S Windows kernel version

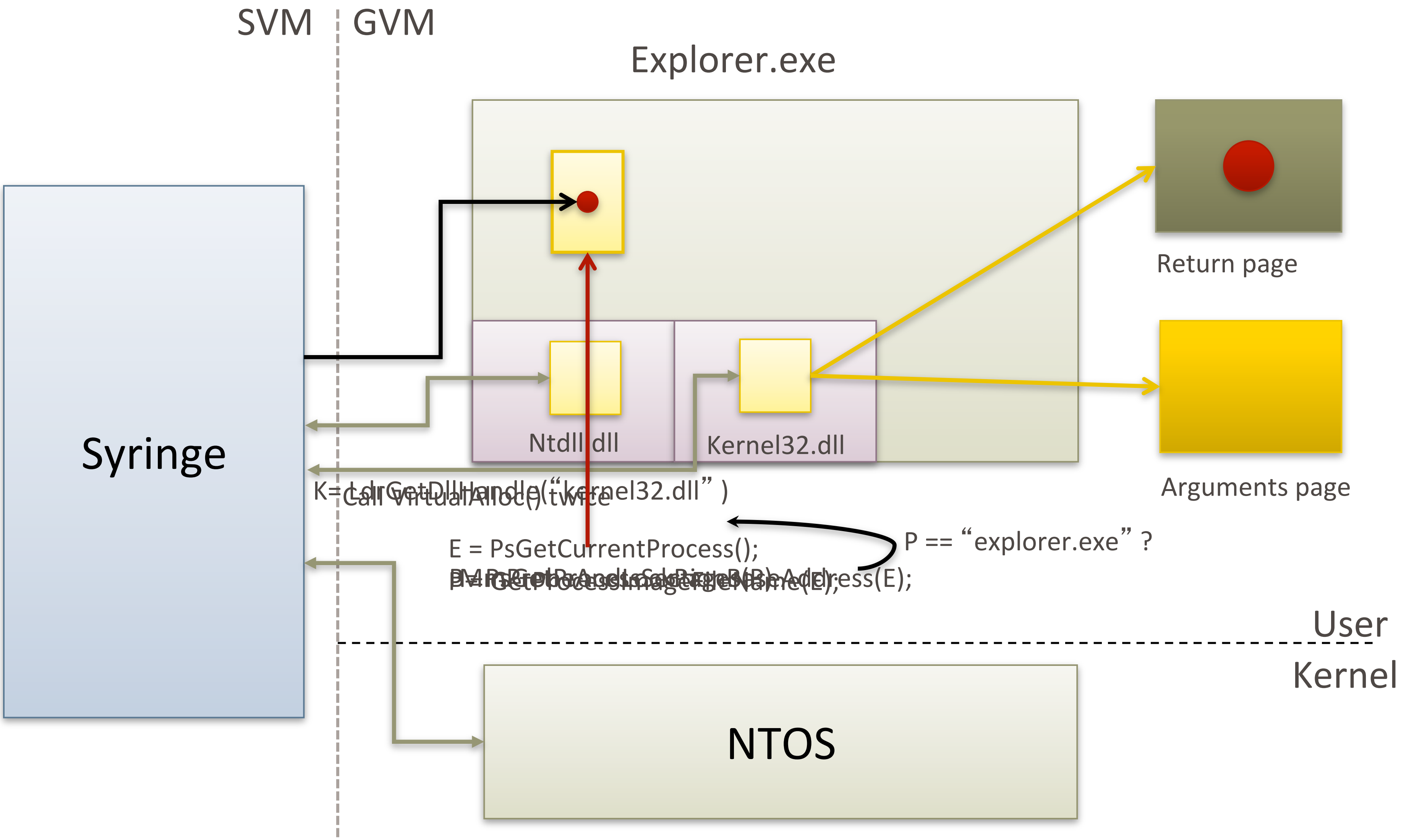
Ntos!PsCreateSystemThread() → Create a kernel-level thread in the GVM

User-Mode FCI

- API at a preferable abstraction level as compared to kernel-mode
- Host process *H* must be chosen
  - Option 1: Resident, high-privilege system process (svchost.exe)
  - Option 2: Resident process that interacts with desktop and runs as a user
  - `Explorer.exe` (Windows Explorer) was chosen for this first prototype

- We have to rely on kernel-mode FCI to bootstrap!
  - Kernel is always running and always available, but Explorer is not
  - User-mode processes are constantly being swapped
  - Need to detect when the currently executing process is Windows Explorer

User-Mode FCI: The Full Process



K=LdrGetDllHandle(“kernel32.dll”)

Call VirtualAlloc() twice

E = PsGetCurrentProcess();

B=PsGetProcessSectionBaseAddress(E);

MmProbeAndLockPages(B)

P= GetProcessImageFileName(E);

P == “explorer.exe” ?

User-Mode FCI: Example APIs

Kernel32!CreateFile()
Kernel32!WriteFile()
Kernel32!CloseFile()

→

File Copying
SVM -> GVM

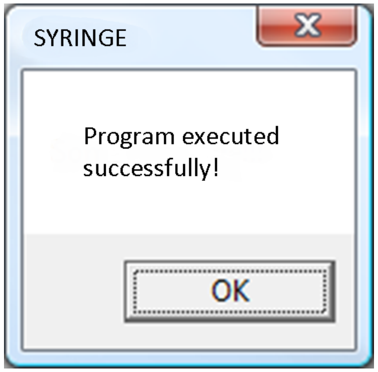
Kernel32!CreateProcess()

→

Execute copied file in GVM

User32!MessageBoxA()

→



Program executed successfully!

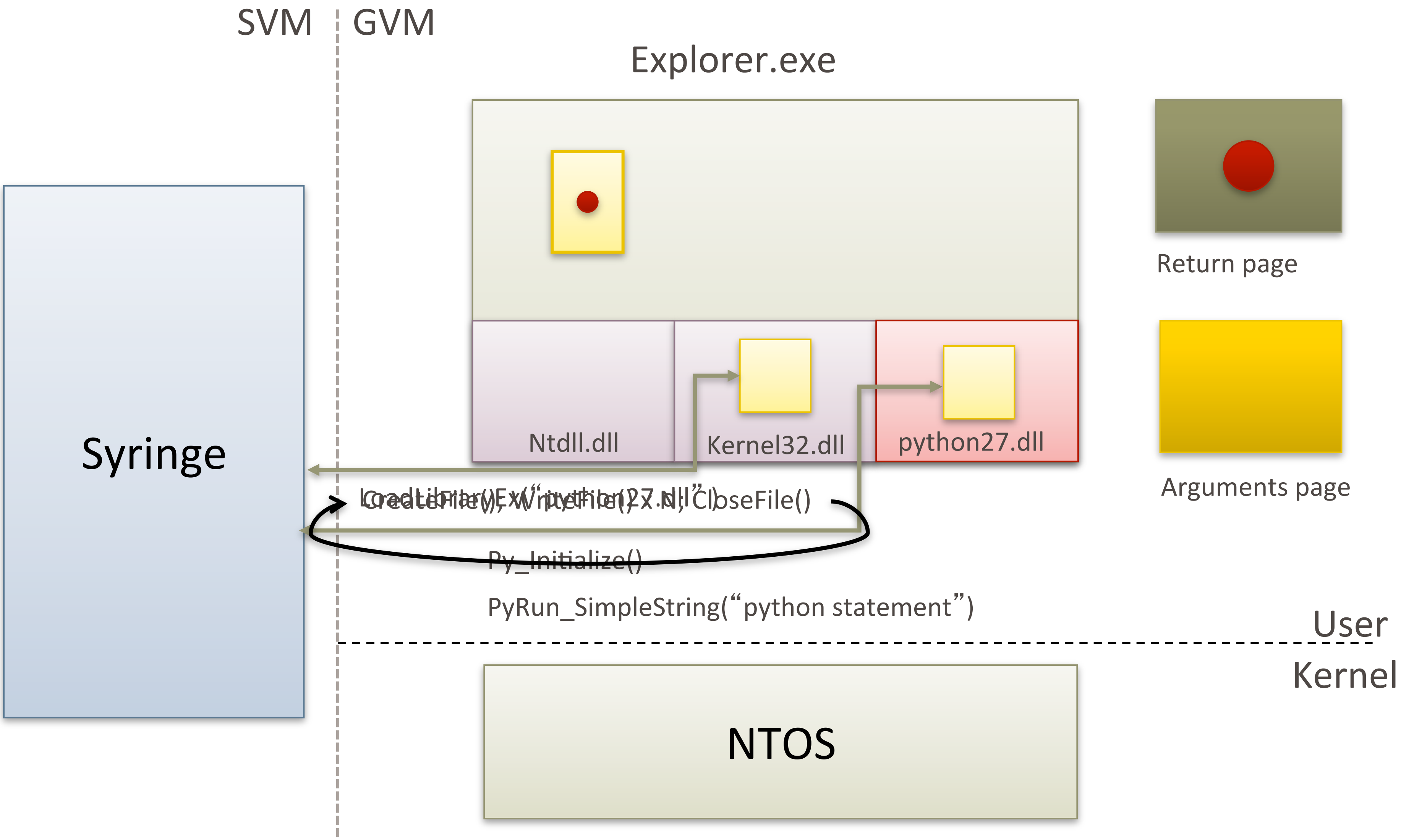
Making FCI OS Independent

- So far we have been focusing on Windows…
- …but FCI is actually an OS-independent technique!

- However, the ***data*** we are injecting and accessing…
  - Function names, addresses, parameters, etc.
- …are OS-dependent.

- Is there some way we could simply abstract all of that?

Inject a cross-platform interpreter (e.g., Python)



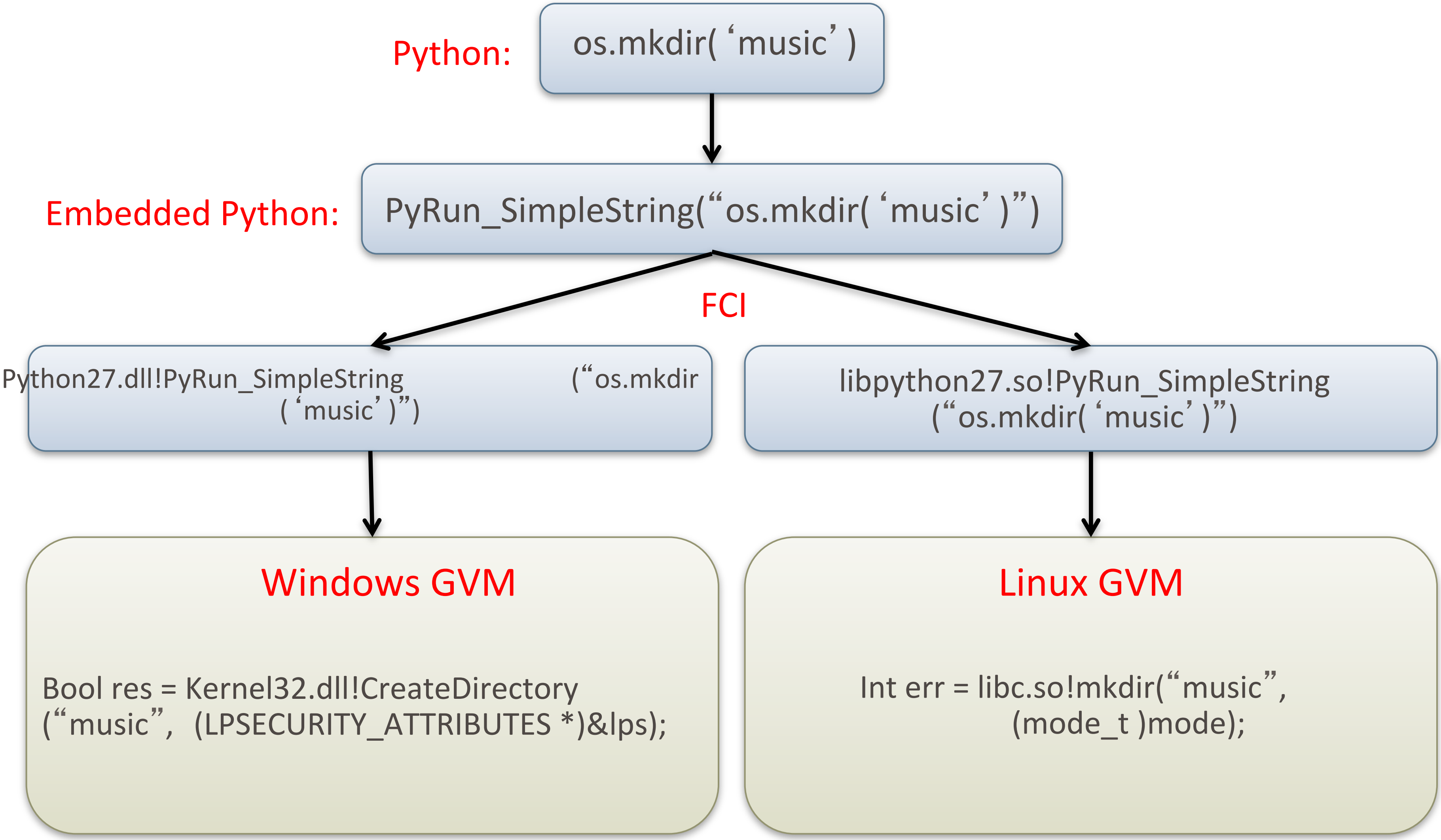
LoadLibraryEx(“python27.dll”)

CreateFile(), WriteFile() x N, CloseFile()

Py_Initialize()

PyRun_SimpleString(“python statement”)

Example



Python: os.mkdir(‘music’)

Embedded Python: PyRun_SimpleString(“os.mkdir(‘music’)”)

FCI

Python27.dll!PyRun_SimpleString (“os.mkdir (‘music’)”)

libpython27.so!PyRun_SimpleString (“os.mkdir(‘music’)”)

Windows GVM

Bool res = Kernel32.dll!CreateDirectory (“music”, (LPSECURITY_ATTRIBUTES *)&lps);

Linux GVM

Int err = libc.so!mkdir(“music”, (mode_t)mode);

Demo / Questions?

Email: mconover@gmail.com