Attacking Host Intrusion Prevention Systems

Eugene Tsyrklevich
eugene@securityarchitects.com

Agenda

- Introduction to HIPS
- Buffer Overflow Protection
- Operating System Protection
- Conclusions
- Demonstration

Introduction to HIPS

- Host Intrusion Prevention Systems are deployed on the end hosts
- Should protect against buffer overflows
- Should protect the underlying operating system
- Should protect against known and unknown attacks

Attack Scenario – Stage 1

- The first step in a typical attack involves gaining remote access to a system
- Usually achieved by means of a remote buffer overflow
- HIPS solution: buffer overflow protection

Attack Scenario – Stage 2

- Once remote access is gained, attackers usually clean the logs, trojan the system and install rootkits
- Achieved by tampering with system logs and binaries and by loading unauthorized malicious code
- HIPS solution: disallow tampering with system files and registry keys and disallow loading of unauthorized code

In reality…

- Buffer overflow protection can be trivially bypassed
- System files and registry keys can be modified
- And kernel code can still be loaded

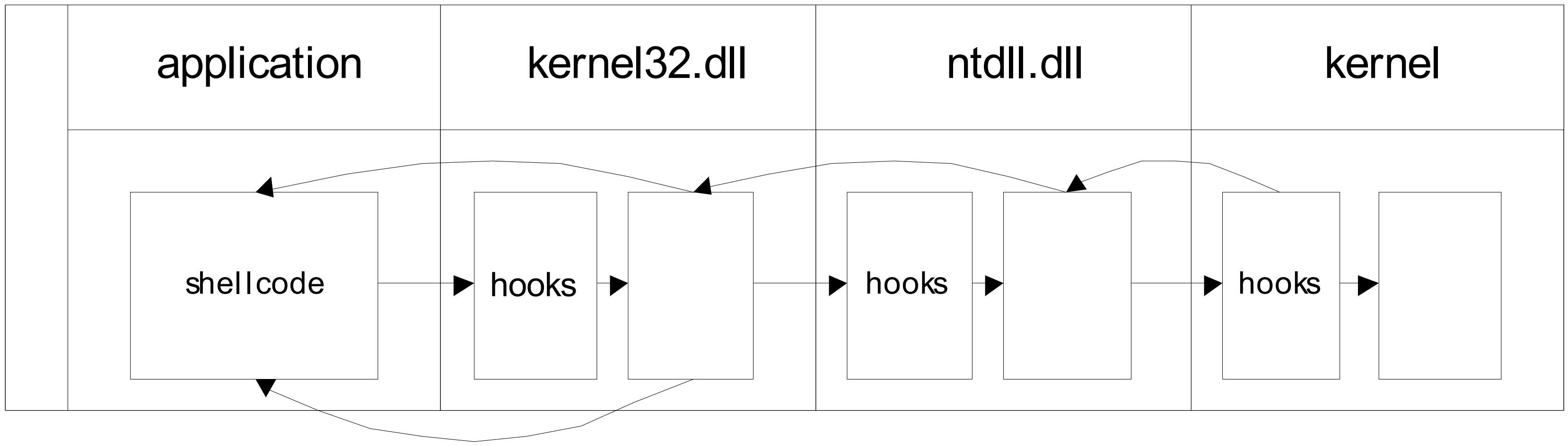
Buffer Overflow Protection

- The majority of existing buffer overflow protection solutions do not actually prevent buffer overflows
- Instead they try to detect when shellcode (attacker's code) begins to execute

Buffer Overflow Protection (2)

- Shellcode detection works by checking whether code is running from a writable page (i.e. stack or heap)
- Shellcode detection can be implemented in
  - Userland or
  - Kernel

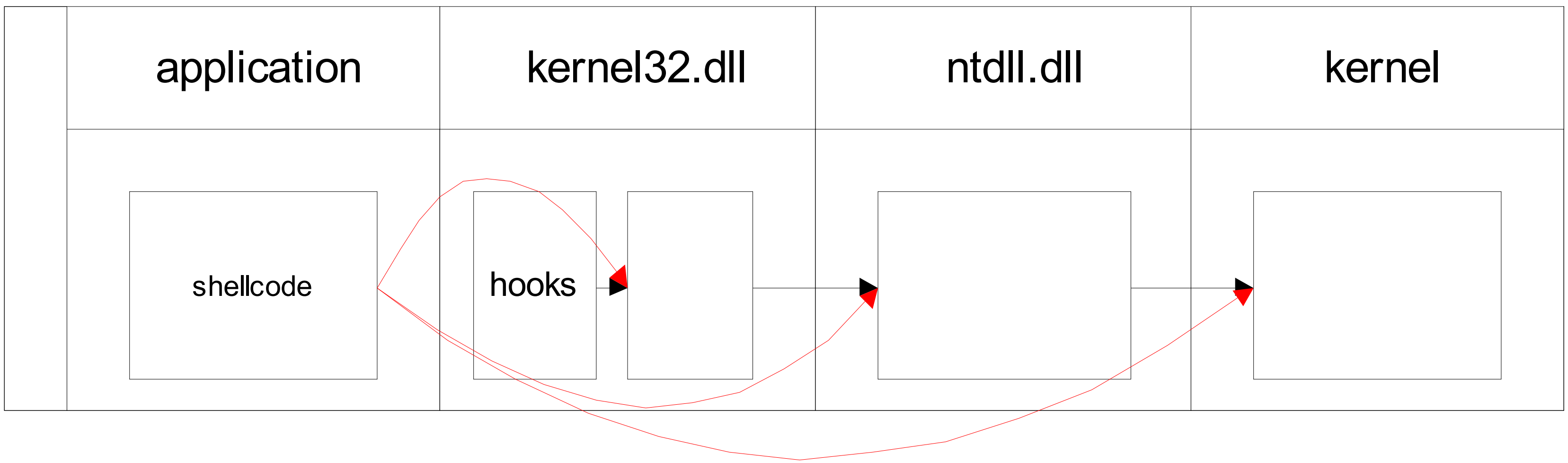
Win32 Example



Win32 Userland Buffer Overflow Protection Code

```
LoadLibraryA: // original function preamble is overwritten by HIPS
    jmp    Kernel32SampleBufferOverflowProtectionHook
    :

void Kernel32SampleBufferOverflowProtectionHook() {
     // retrieve the return address from stack
    _asm mov ReturnAddress, [esp]

    if (IsAddressOnWritablePage( ReturnAddress ))
          LogAndTerminateProcess();

    ReturnToTheHookedAPI();
```

Bypassing Userland Hooks



- It is possible to bypass kernel32.dll hooks and call other entry points directly!

Bypassing Userland Hooks Example

- Normal shellcode

```
void shellcode()
{
    LoadLibrary("library.dll");            // call kernel32.dll which
}                                          // will eventually call ntdll.dll
```

- "Stealth" shellcode

```
void shellcode()
{
    LdrLoadDll(… "library.dll" …);         // call ntdll.dll directly
}
```

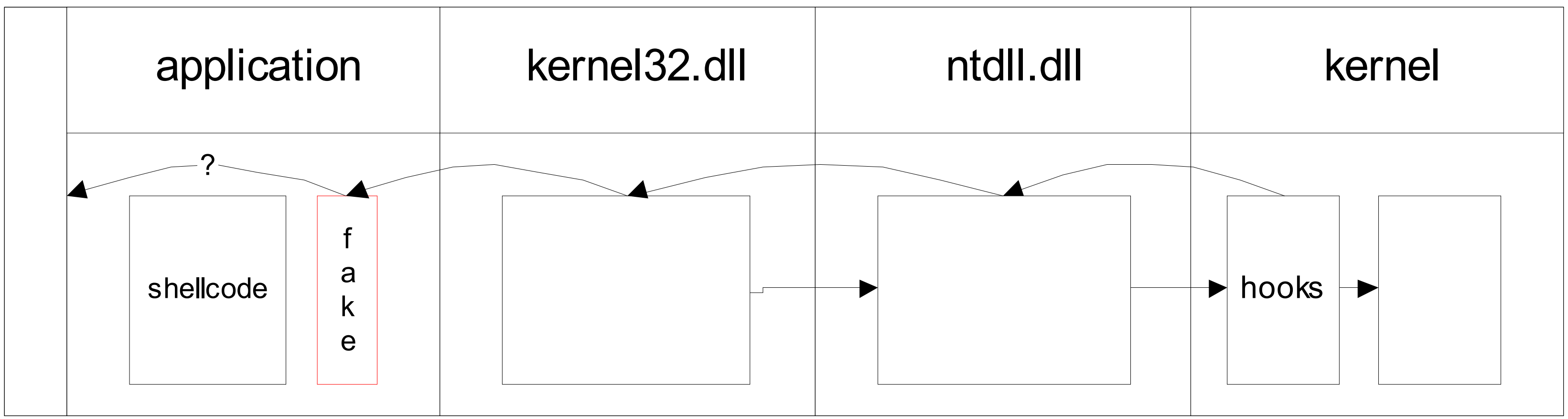
Attacking Userland Hooks

- Userland hooks run with the same privileges as the shellcode
- Therefore, shellcode, in addition to simply bypassing the hooks, can attack the protection mechanism directly
- This applies not only to buffer overflow protection but also to all security mechanisms implemented in userland

Attacking Userland Hooks Example

```
void shellcode()
{
    // bypass GetProcAddress() hook
    LoadLibraryAddress =
    ShellCodeCopyOfGetProcAddress("LoadLibraryA");

    // overwrite LoadLibraryA() hook with the original function preamble
    memcpy(LoadLibraryAddress, LoadLibraryAPreamble, 5);

    // call “cleansed” LoadLibrary()
    LoadLibraryAddress();
}
```

Bypassing Kernel Hooks

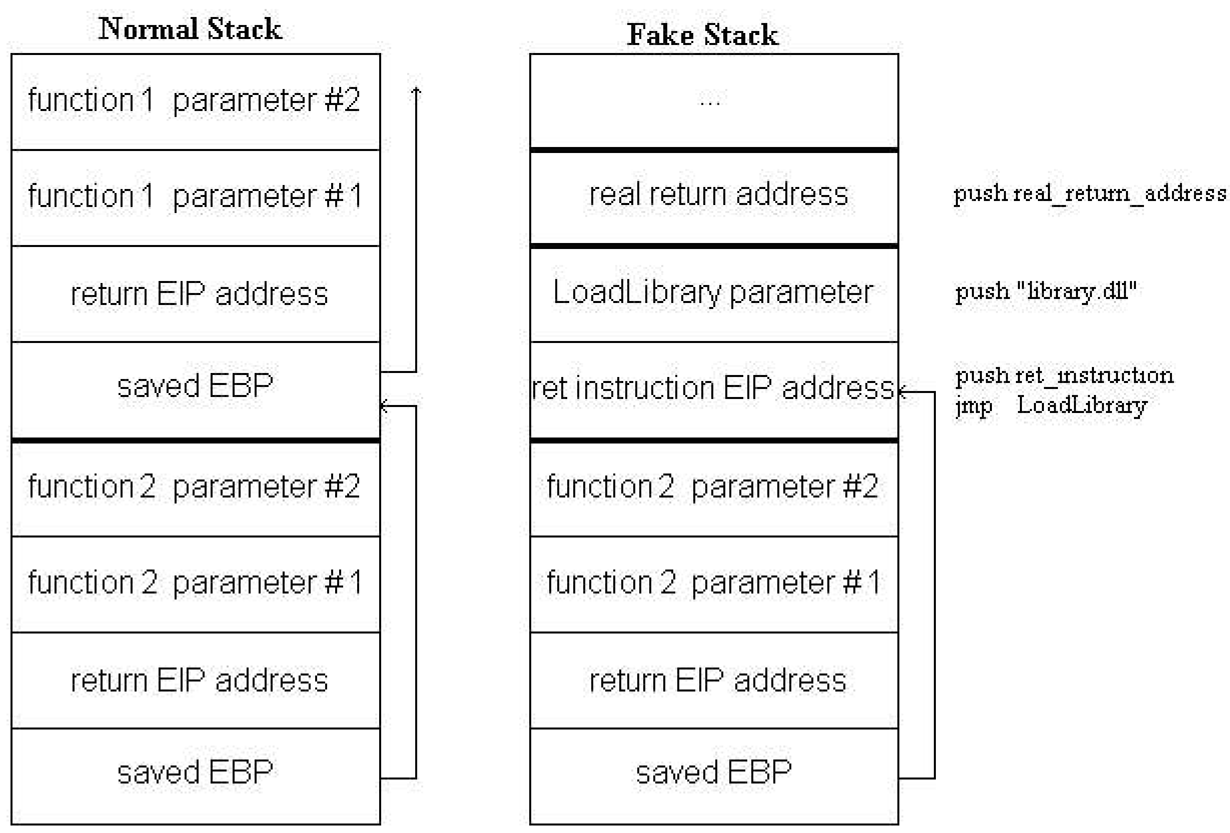


- Create a fake stack frame without the EBP register and with a return address pointing to a non-writable segment

Bypassing Kernel Hooks Example

```
    // LoadLibrary("library.dll")
    push            real_return_address

    push            "library.dll"

    // fake a "call LoadLibrary" call with a fake return address
    push            ret_instruction_in_readonly_segment
    jmp             LoadLibrary
real_return_address:
            :
ret_instruction_in_readonly_segment:
            ret
```

Bypassing Kernel Hooks Example (2)



Buffer Overflow Protection Summary

- Hard to implement in a secure manner
- Even harder to implement on a closed source operating system
- The majority of buffer overflow protection solutions are simply designed to detect shellcode
- Can be easily bypassed by attackers

Operating System Protection

- Operating system protection involves protecting the integrity of system files and registry keys
- Operating system protection also disallows the loading of arbitrary code
- Similar to buffer overflow protection, operating system protection can be implemented in
  - Userland or
  - Kernel

Userland OS Protection

- Userland protection code runs with the same privileges as the shellcode
- Win32 SAFER appears to be implemented this way
- Completely ineffective against malicious code that has already begun to execute

Kernel OS Protection

- Kernel code runs with different privileges than userland
- Has complete control over the entire system
- Hard to attack directly
- But can still be evaded (if not implemented properly)

Bypassing Operating System Protection

- Some HIPS implementations can be completely bypassed by using symbolic links
- HIPS might be protecting c:\windows\system32\drivers*
- But is it protecting x:\drivers* ?

Bypassing Operating System Protection Example

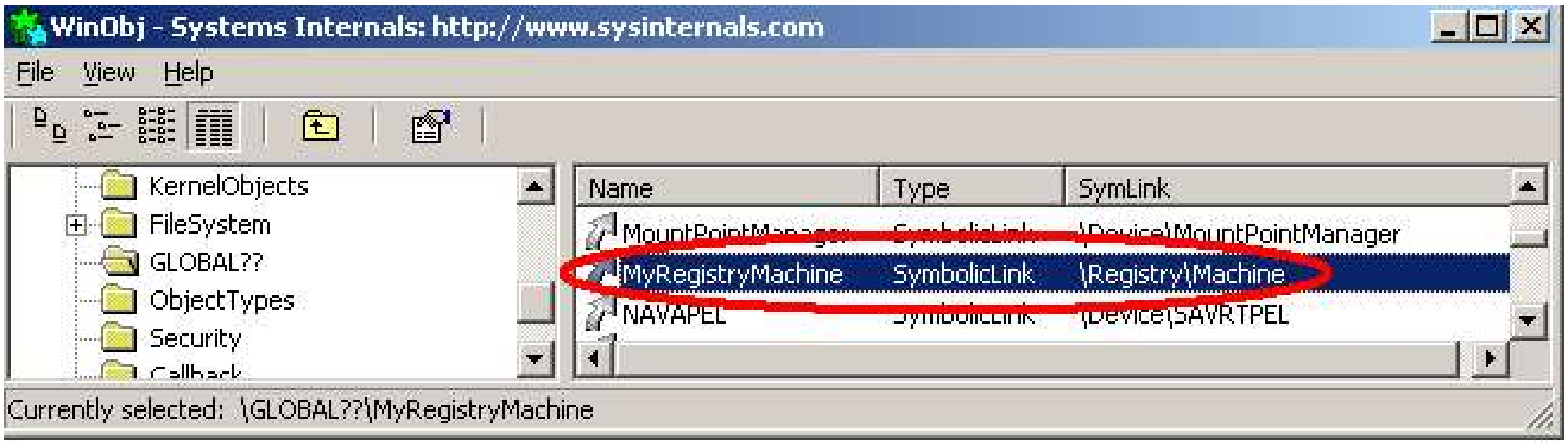
C:\WINDOWS\System32\cmd.exe

```
C:\>dir c:\windows\system32\drivers\ipfltdrv.sys
 Volume in drive C has no label.
 Volume Serial Number is AC03-0FBE

 Directory of c:\windows\system32\drivers

29/08/2002  13:00            32,896 ipfltdrv.sys
               1 File(s)         32,896 bytes
               0 Dir(s)   1,259,008,000 bytes free

C:\>subst x: c:\windows\system32

C:\>dir x:\drivers\ipfltdrv.sys
 Volume in drive X has no label.
 Volume Serial Number is AC03-0FBE

 Directory of x:\drivers

29/08/2002  13:00            32,896 ipfltdrv.sys
               1 File(s)         32,896 bytes
               0 Dir(s)   1,259,008,000 bytes free

C:\>_
```

Bypassing Operating System Protection (2)

- Alternatively, HIPS might be protecting \Registry\Machine\System*
- But is it protecting \MyRegistryMachine\System* ?
- NtCreateSymbolicLinkObject() can be used to create symbolic links in kernel namespace

Bypassing Operating System Protection Example (2)



- \MyRegistryMachine\System = \Registry\Machine\System.

Kernel Code Loading Interfaces

- A well-known and well understood interface: Service Control Manager (SCM) API
- A less known interface: ZwLoadDriver()
- A little known interface: ZwSetSystemInformation()
  - SystemLoadAndCallImage
  - SystemLoadImage

Bypassing Kernel Code Loading Restriction

- Use a little known interface such as ZwSetSystemInformation()
- Inject code by directly modifying kernel memory (\Device\PhysicalMemory or is it \MyPhysicalMemory? :)
- Exploit a kernel overflow

Bypassing Kernel Code Loading Restriction

- If a trusted system process is still allowed to load kernel drivers, use DLL injection to inject userland code into the trusted process and then load a malicious kernel driver
- Modify an existing kernel driver on disk

Operating System Protection Summary

- HIPS are designed to protect operating system files and registry keys, as well as to disallow the loading of unauthorized code.
- Similar to buffer overflow protection, userland based implementations cannot protect against malicious code that is executing with the same privileges
- Kernel based implementations are a lot more robust, but can still be evaded by modifying different system namespaces

Conclusion

- HIPS technology has a promising future
- There are a lot of attack vectors and missing just one could completely compromise the security and integrity of the system
- The majority of current HIPS implementations suffer from a variety of security flaws
- The technology needs time to mature

Thank You

Thanks!

http://www.securityarchitects.com/

eugene@securityarchitects.com

Demonstration

- Live Demo