

Eugene Tsyркlevich



BLACK HAT BRIEFINGS

Ozone HIPS: Unbreakable Windows

Windows is the number one target on the Internet today. It takes less than 5 minutes for an unpatched Windows machine, connected to the Internet, to get owned. Yet the most prevalent security practices still consist of running anti-viruses and constant patching.

This presentation introduces a new tool, called Ozone, that is designed to protect against most of the commonly exploited attack vectors. To protect against the most common of these, buffer overflows, Ozone uses an address space randomization technique. In addition, Ozone runs all processes in a sandbox that severely limits what a compromised process is allowed to do. Finally, Ozone protects itself and the underlying operating system against further attacks.

Eugene Tsyркlevich has an extensive security background ranging from designing and implementing Host Intrusion Prevention Systems to training people in research, corporate, and military environments. Eugene has presented his research at a number of security conferences including Usenix Security, BlackHat Europe and BlackHat USA. Eugene holds both a Bachelor and a Masters degree in Computer Science from the University of California, San Diego.

Ozone HIPS: Unbreakable Windows

Eugene Tsyrklevich

Agenda

- Memory Protection
- Process Protection
- System/OS Protection
- Ozone's self protection
- Demonstration

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

Solution: buffer overflow protection + process sandboxing

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

Solution: disallow tampering with system resources and/or disallow loading of unauthorized code

Memory Protection

The memory protection layer is responsible for guarding against attacks that hijack execution by corrupting memory

These include

- Buffer overflows (stack and heap based)
- Format bugs
- Function pointer overwrites
- Other data corruption

Exploiting Memory Bugs

The majority of memory corruption attacks involve

1. Injecting a malicious payload
2. Transferring control to it

We cannot prevent payload injection but can prevent attackers from gaining control

Stopping Memory Exploits

Writable page execution (Entercept)

Canary / compiler based solutions
(StackGuard, ProPolice)

Address Space Randomization (PaX, Ozone)

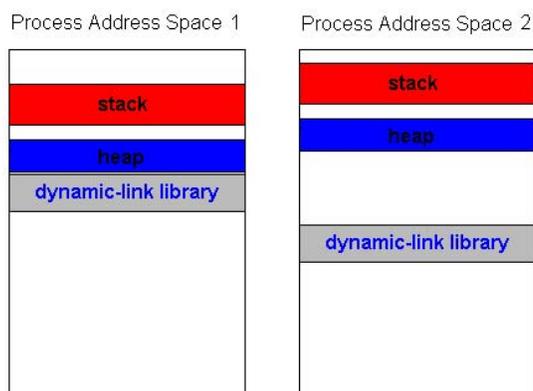
Many, many others

Address Space Randomization

Address Space Randomization involves randomizing the base addresses of various data structures, such as stack, heap, libraries, code segment, etc

As the address space layout of each instance of an application is different from the next, all exploits depending on static addresses will stop working

ASR Illustration



ASR Implementation Details

Stack and heap randomization

- Has 0% overhead
- Provides ~16 bits of entropy
- Does not break any sane applications

Runtime DLL randomization

- Has an upfront CPU cost (time to relocate) plus some memory overhead (DLL code is not shared any longer)
- Provides ~14 bits of entropy
- Might break certain apps

ASR Implementation Challenges

Win32 loader prevents kernel32.dll (and user32.dll) from being rebased because BaseProcessEntry address is copied from one process to another during CreateProcess()

BaseProcessEntry() address can be adjusted at runtime and the loader can be tricked

Ntdll.dll is loaded at bootup time and then mapped into each process

Ntdll.dll can be statically rebased. Runtime solutions are being investigated.

ASR Implementation Challenges (2)

As an optimization, Win32 compilers do not include relocation information for executables.

This means that .text segments cannot easily be relocated.

Possible solutions:

- Force compilers to include relocation info

- Relocate and rewrite the binary manually

ASR turns buffer overflows into Denial of Service which might not be acceptable for all applications

ASR Conclusion

ASR involves randomizing the base addresses of all memory regions, including stack, heap, DLL and executable

ASR does not provide a 100% guarantee against all memory based attacks

ASR can provide additional security with little overhead

ASR stack and heap randomization (at least) should be included in a stock OS

Process Protection

To provide further protection against a variety of attack vectors, the process protection layer is responsible for executing all processes inside the sandbox

The sandbox enforces process rules from kernel mode and cannot be bypassed

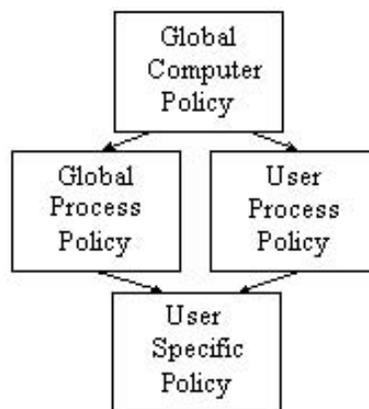
The sandbox mediates access to all system resources (~20 types), which a process might need to access

Policy Engine

Sandboxes are described by policy files

A policy file is a collection of rules specified in a text file

Policy Hierarchy



Policy Example

```
# run iTunes & Skype concurrently with SoftICE
file_all: name eq "\Device\Siwvid" then deny
registry_read: name match "**\secret" then quietdeny
dll_load: log
driver_load: deny
process_execute: name eq "ok.exe" then permit
network_tcpconnect: name eq "127.0.0.1" then permit
# act as a personal firewall
network_all: ask
```

Policy Types

Deny all policy

```
process_execute: name eq "ok.exe" then permit
process_execute: deny
```

Allow all policy

```
process_execute: name eq "bad.exe" then deny
process_execute: permit
```

Policy Creation

Manual

Suitable for allow all policies where only known malicious resources are denied access to

As the name implies, an administrator manually specifies all the necessary rules

Automatic

Suitable for deny all policies where only a known good subset of resources is allowed access to

Automatically generated by monitoring process behavior

Process Protection Challenges

Svchost.exe/dllhost.exe host applications

IE policy: protection_all: off

Finally, spot 2 issues in the screenshot below of Acrobat.exe's loaded DLL list that make sandboxing on Windows such a challenging task..

Modules						
Name	Address	Path	Order	Version	Program	
☒ samlib.dll	71BF0000-71C01000	C:\WINDOWS\system32\samlib.dll	94	5.01.2600...	[3220]	acrobat.exe: Native
☒ davclnt.dll	75F70000-75F79000	C:\WINDOWS\system32\davclnt.dll	95	5.01.2600.0	[3220]	acrobat.exe: Native
☒ setupapi.dll	76670000-76757000	C:\WINDOWS\system32\setupapi.dll	96	5.01.2600...	[3220]	acrobat.exe: Native
☒ ntsshrui.dll	76990000-769B4000	C:\WINDOWS\system32\ntshrui.dll	97	5.01.2600...	[3220]	acrobat.exe: Native
☒ atl.dll	76B20000-76B35000	C:\WINDOWS\system32\atl.dll	98	3.00.9435.0	[3220]	acrobat.exe: Native
☒ idle.dll	60300000-60307000	C:\Program Files\Yahoo!\Messenger\idle.dll	99	1.00.0.2	[3220]	acrobat.exe: Native
☒ msvcr71.dll	7C340000-7C396000	C:\Program Files\Yahoo!\Messenger\msvcr71.dll	100	7.10.3052.4	[3220]	acrobat.exe: Native

Protection Protection Conclusion

All processes are executed inside a sandbox, which is enforced from kernel mode and cannot be bypassed

Sandbox mechanism controls access to all named system resources

Sandboxes are described using text files which list access rules to all specified resources

System Protection

The system protection layer is designed to harden the underlying operating system

This is achieved by protecting the integrity of static and dynamic core OS binaries & registry keys

In addition, malicious or potentially exploitable features such as debugging, DOS16 emulation, keyboard loggers and kernel driver loading can be disallowed

Finally, access to removable media can also be restricted

System Protection Policy Example

protection_debugging: on
protection_dos16: on
protection_keyboard: on
time_change: log
token_modify: permit
media_access: readonly

System Protection Challenges

Legitimate applications require loading of
kernel code drivers
Windows security patches regularly
overwrite/update core OS resources

System Protection Conclusion

The system protection layer is responsible for protecting the underlying operating system

This is achieved by protecting the core file and registry resources

In addition, dangerous functionality such as debugging, dos16 emulation, keyboard logging and removable media access, can all be restricted

Self Protection

As the name states, the self protection layer is responsible for protecting Ozone itself

To achieve this, all the core components such as userland services and kernel mode drivers cannot be stopped or unloaded

In addition, kernel code drivers cannot be loaded to prevent further attacks

All the security rules are enforced from the kernel mode and cannot be bypassed by userland processes

Conclusions

Modern systems are subject to a number of powerful attacks which no single mechanism/technique can prevent

To address this issue, Ozone was designed with a “security in depth” principle in mind

Ozone consists of multiple “layers” designed to stop a variety of attack vectors ranging from remote memory based attacks to local privilege escalation

Conclusions (2)

The memory layer uses an ASR technique to prevent memory based attacks

The protection layer uses a sandbox mechanism to protect against a variety of attack vectors

The system protection layer hardens the underlying operating system

Ozone is also designed to protect itself against malicious behaviour

BLACK HAT BRIEFINGS

