Application Sandbox Limitations

Presenting a better way to defeat advanced threats
Introduction
As adversaries have become more sophisticated the “detect to protect” paradigm of endpoint security has been widely recognized as inadequate. Polymorphism makes it easy for an attacker to fool signature-based detection, and anomaly and behavioral analysis requires near-perfect models of “good” or “bad” behavior. Modern attacks target specific organizations or individuals, are executed with patience and care, and may involve direct interaction by a human attacker via a Remote Access Tool (or RAT), making the attack impossible to detect.

As a result, there is great interest in proactive methods to block attacks, for example:

- **White-listing** aims to prevent any code that is not explicitly allowed (white-listed) from executing. It is effective in highly restrictive environments but is vulnerable to compromise of the root of trust. It is also impractical if the user has administrative privileges or accesses web-based interpreted code, including Java.

- Alternatively, many applications that interact with untrustworthy content include a Sandbox (including web browsers, document and content renderers, and the Java VM) to try to contain an attacker without overly compromising the user's experience. A sandbox attempts to mitigate the inherent complexity (and therefore vulnerability) of:
  - large applications, in particular those that run untrustworthy 3rd party code, and
  - modern operating systems that consist of tens of millions of lines of code.

This paper focuses on application sandboxing, and identifies fundamental protection limitations of this approach through architectural analysis and practical experimentation. Our key findings are

- Every sandbox can be compromised, due to design or coding flaws in the sandbox or the operating system it aims to protect.
- Microsoft® Windows has many kernel flaws that cannot be protected using a sandbox.
- There has been a rapid increase in attacks that can escape any sandbox.

We conclude that a substantially more resilient approach is required to protect vulnerable systems. **Bromium micro-virtualization** is one such technique that hardware-isolates each untrustworthy task using CPU features for virtualization, offering protection that is many orders of magnitude higher than possible with any existing endpoint security mechanism today.

The Inherent Weakness of Application Sandboxes
Application sandboxes have been widely adopted by vendors struggling to contain repeated exploits. Their success is dependent both on the architecture and the implementation quality of the sandbox itself – with different vendors achieving markedly different results – and also on something beyond the control of the application developer, namely the architecture and implementation quality of the operating system that the sandbox aims to protect (in our case, Windows).

In this paper we focus on a key architectural weakness of application sandboxes, namely their reliance on Windows kernel security. The Windows kernel presents a broad attack surface to a skilled attacker, and a successful exploit against the kernel will breach the sandbox. This fact is hardly news – most good papers about various sandbox implementations mention the possibility of a kernel-based attack. However, we assert that this attack vector should be considered as a very real threat, at least as likely to be
exploited as other possible vectors (e.g. bugs in the sandbox implementation). We have developed proof-of-concept exploits that use this technique to compromise four different commercial sandboxes when run on a version of Windows with un-patched kernel vulnerabilities.

There is empirical evidence that there are relatively few vulnerabilities in major vendor sandboxes. By comparison, Windows kernel vulnerabilities are plentiful and new ones frequently found. In 2012, 25 CVE entries were allocated for Windows kernel-mode components that permitted privilege escalation to kernel mode. Microsoft security bulletins for February 2013 lists 30 CVE vulnerabilities in the kernel mode component win32k.sys alone. Exploitation of kernel vulnerabilities is more complex than user-mode applications and requires more knowledge of the OS implementation. This is presumably why malware authors have not yet utilized this attack vector widely (with Duqu [1] a notable exception). Nonetheless, the growing number of Windows kernel vulnerabilities announced each month confirms our view that this attack vector should be considered dominant, and that it will be more actively exploited in the future to defeat sandboxes.

Each new vulnerability in the Windows kernel offers attackers a new opportunity to attack the application sandbox. In the rest of this paper we study two key sandbox architectures, describing exploits that take advantage of kernel vulnerabilities to defeat containment.

Type 1: The “OS Enhancement” Sandbox

The Type 1 sandbox is typically intended by the vendor to be a general purpose sandbox that could be used to contain any (or most) applications. Examples include Sandboxie, Trustware Bufferzon, and Invincea.

Architecture

In this type of sandbox the vendor implements a custom kernel driver that modifies the behavior of Windows for a particular sandboxed application to provide additional protection. By way of example:

› If a sandboxed application tries to terminate a non-sandboxed process, it will be blocked.

› If a sandboxed application tries to write to a protected file (listed in the sandbox configuration), then the driver might transparently create a copy of the file that the sandboxed application can write to, while the original file is untouched. Such a copy-on-write mechanism allows the sandboxed application to work unchanged, while preventing any damage to the rest of the system.

Architectural Flaws

In this approach the sandbox design focuses on preventing access to protected resources. If well implemented, this goal can be achieved. However, independent of the completeness of the sandbox itself, this architecture still allows access to most kernel functionality, and therefore to kernel vulnerabilities. This is due to the general purpose nature of this type of sandbox, in which an application may need access to many kernel interfaces. Simply protecting access to key resources such as files, printers, devices and so on does not offer adequate protection. Once an attacker can compromise the kernel via interfaces that are left unprotected, he can completely bypass the sandbox, since it has no way to protect itself from code running at the same (escalated) privilege level.
Example: Sandboxie

Sandboxie [2] is a popular sandbox toolkit of reasonable implementation quality, independently managed by Ronen Tzur. It is widely used for malware analysis and controlled application execution. Unfortunately the Sandboxie documentation does not describe the system architecture or implementation. Figure 1 (drawn from an unrelated site[3]) shows the main components of Sandboxie. (This documentation may be incomplete or outdated). Due to the differences in x86_32 and x86_64 Windows kernels, Sandboxie's method for hooking crucial operations differs between these platforms. The documentation states that the “experimental” Sandboxie mode on x86_64 provides nearly the same level of protection as for x86_32.

We tested an exploit for CVE-2012-0217 (misleadingly named “User Mode Scheduler Memory Corruption” in Microsoft bulletin MS12-042] in using Sandboxie on an x86_64 platform running Windows 7 SP1. This vulnerability is caused by not sanitizing the return address of a system call. A non-canonical return address results in an unexpected exception being raised, that is in turn handled incorrectly. Our exploit succeeded, allowing us to run arbitrary code in kernel mode. A slight twist is that kernel shell code that steals the SYSTEM access token is not particularly useful to the attacker: Although the attacker gains SYSTEM rights in his user mode process, this process is still confined by Sandboxie (and, for example, an attempt to kill a non-sandboxed process from the SYSTEM shell fails). An attacker needs to do some extra work while in kernel mode, either:

1) Disabling the Sandboxie driver (via uninstall hooks, or overwriting the driver code)
2) Migrating to another process that runs outside of the Sandboxie container, before continuing the attack

We chose the second method, because it is more generic. The required steps are:

1) Allocate kernel memory for an exploit_syscall_handler() function
2) Hook all system calls via overwriting LSTAR MSR, (LSTAR := exploit_syscall_handler)
3) When the exploit_syscall_handler() function detects that it is run in a context of a process running outside of the sandbox, it injects arbitrary shell code into this process

The result is that after the exploit is run within the sandbox, the attacker can execute his code in the context of any process. A careful reader may recognize that this procedure is similar to how remote kernel exploits (that often initially execute with raised IRQL) migrate to a stable user mode environment.

To summarize: we have verified that by exploiting CVE-2012-0217, a Sandboxie-confine process can completely bypass Sandboxie protection.

Example: BufferZone Pro [4]

This popular sandbox is similar to Sandboxie. Naturally many implementation details are different – for example by default, it prevents read access to selected directories (to prevent data theft). We have validated that he same CVE-2012-0217 shell code-migrating exploit can be used to fully bypass BufferZone Pro protection running on unpatched Windows 7 SP1 on an x86_64 platform.
Type 2: The “Master/Slave” Sandbox Architecture
This approach can only be used by application vendors seeking to contain untrusted execution, because the application must be re-architected into two independent processes. The first (the Slave), runs with very low privilege (using standard OS access control facilities). It is allowed to access almost no kernel resources. Whenever the Slave needs access to a particular resource (for example to read a file), it sends a message to the other process (the Master) via an IPC channel. The Master validates the request and if it is legitimate, performs the required operation on behalf of the Slave before returning any results, again via IPC.

The idea is that the vast majority of the application code stays in the Slave. If the Slave is compromised, the OS will prevent it from directly accessing any valuable resources. The Master’s codebase is small and therefore ought to have few bugs, making it difficult for a compromised Slave to further compromise the Master.

The Chrome sandbox
The Chrome sandbox is a superior implementation used by both the Chrome browser and Adobe®, with some implementation differences. The accompanying diagram, drawn from documentation for the Chrome sandbox [5], shows the key components:

The Slave process indeed runs with low privileges, namely:

- Restricted token
- Job object
- Desktop object
- Lowered integrity level

According to the Chrome documentation, this privilege level permits very few operations:

- Access to mounted FAT or FAT32 volumes
- TCP/IP communication
- No other resources (including, for example, files on an NTFS file system) can be accessed directly.

The following snapshot of the ProcMon process monitor illustrates the protection architecture of the Chrome sandbox implementation in Adobe Acrobat.
In this picture, process 2856 is the Slave. It has tried to access a file, and the operating system has denied this request. The Slave sends a message “open a file for me please” to the Master (unfortunately, the Process Monitor does not show named pipe activity, so the IPC cannot be seen). The Master (process 2808) opens the file and duplicates the file handle into the Slave process. The Slave can then access the file using the receive handle.

Both the Chrome browser and Adobe Acrobat Reader use the Chrome sandbox but the actual implementation details are slightly different. The following screenshots show Adobe Reader Slave access token details (Left) and the Chrome browser Slave (named “renderer” in the Chrome documentation) access token details (Right):

![Adobe Reader Slave Access Tokens](image1)

![Chrome Browser Slave Access Tokens](image2)

One immediately visible difference is the integrity level – “low” in Acrobat Reader, versus “untrusted” in the Chrome browser. Notably, the Chrome browser Slave is not permitted by the OS to access FAT32 file systems (contradicting the Chrome documentation), while the Adobe Reader Slave can freely do so. There are other key differences in the policies enforced by the two Masters. In comparison with the Chrome browser, the Adobe Reader Master permits access to many more registry entries and many more kinds of file system access (For example, the Adobe Reader Master can open any Dynamic Link Library file read-only).

In addition, while it is easy to use TCP/IP networking in the Adobe Slave (it is possible to simply load the winsock library), this is not the case in the Chrome Slave. We have not yet determined whether the Chrome Slave has sufficient privileges to create TCP/IP socket but we have verified that it can open the \Device\Afd\Endpoint file, which is a crucial step in the process of creating a socket. An ability to connect to arbitrary local ports or services on a corporate network constitutes a possible escalation.

Architectural Flaws
Both the Chrome browser and Adobe Reader sandboxes have good vulnerability records. The first case of an Adobe Reader sandbox vulnerability exploited in the wild was reported in February 2013 [6] (the escape was possible because of a bug in the Master). Similarly, attacks against the Chrome browser Master are rare ([7], [8]). It is fair to conclude that if vulnerabilities in the Master were the only concern, then due to the fact that the Master has a small code base, this type of sandbox would provide effective isolation.
Unfortunately, however, this type of sandbox does not adequately reduce the exposure of the underlying kernel, even though some key operations that may be necessary for an exploit to succeed are not allowed. For example:

- The job object prevents creation of processes
- Registry keys normally accessible by unprivileged users are not available

On Linux it is possible to substantially reduce the kernel interface exposure to the slave. For example, using the seccomp-bpf syscall filter [9] (used by Chrome on Linux), it is possible to significantly reduce access to functionality that is not normally needed by the Slave. However the remaining interfaces could still contain exploitable vulnerabilities.

Unfortunately, equivalent functionality is not available on Windows, leaving the kernel vulnerable to exploitation across a broad interface. The largest vulnerable kernel component is the graphics subsystem implemented in win32k.sys, most of which is accessible by the Chrome Slave. Although many other operations of use to an attacker are prohibited (including interaction with windows of other processes) the sandboxed Slave process can still create windows and exploit bugs in the graphics subsystem.

Example: CVE-2012-2897

This vulnerability (patched in MS12-075) is related to a bug in true type font handling in the Windows kernel, namely an integer overflow in the kernel code responsible for parsing the cmap true type font table. Triggering the vulnerability is simple enough:

Simply opening a carefully crafted web page using the Chrome browser on a vulnerable Windows system will suffice. The following screenshot shows a kernel debugger back-trace (Note the red-circled faulting module):

In this case, it is not even necessary to compromise the renderer first – it is enough to simply pass it a crafted .ttf file and reference in a web page. However, in order to achieve code execution reliably, more control over the kernel memory layout is needed; for this, the ability to run arbitrary code in the renderer is very helpful.
Example: CVE-2011-3042

This vulnerability (patched in MS11-087) is also related to true type font parsing. The root cause is an insecure destination buffer address calculation, when processing a malformed compound bitmap stored in the true type font file. It was exploited in the wild by Duqu[1], using malformed MS Office documents. We have experimentally verified that on Windows 7 SP1 a compromised Chrome Slave is able to exploit this vulnerability and achieve arbitrary code execution in kernel mode. The typical token-stealing shell code is all an attacker needs in this case. The following screenshot shows the state of the Adobe Reader Slave after successful exploitation:

The following screenshot shows the state of the Chrome browser Slave after successful exploitation:

It can be seen that after the exploit has succeeded, the Slaves run with highest user mode privilege, and are capable of taking full control over the OS.
Example: CVE-2011-2018
Graphics subsystem vulnerabilities are not the only vulnerabilities that can be triggered from a compromised Slave. CVE-2011-2018, misleadingly referred to as “Windows Kernel Exception handler Vulnerability” in MS11-098, results from treating a particular segment selector value as special, and wrongly assuming that it is not possible modify it from user-mode code. This triggers an unexpected code path when returning from a system call when this special value is seen. This can be used to trivially cause a BSOD when exploited from within the Chrome browser or Adobe Reader sandbox:

A Note on Methodology
In our experiments with type 2 sandboxes, we did not want to spend time developing exploits that are able to get initial code execution in the context of Slave. After all, the core sandbox concept is that we expect vulnerabilities in the Slave to be present and exploitable. The change-logs of both Chrome browser and Adobe Reader confirm there have been a considerable number of coding errors fixed in the Slaves in each software release. Assuming we could exploit the Slave, we have focused on the second stage of an exploit – when an attacker is able to run arbitrary code in the context of Slave and tries to bypass the sandbox. We ran the kernel exploits in the context of the Slaves by manually copying the exploit code via WriteProcessMemory, and triggered them using CreateRemoteThread. It is straightforward to see how shell code triggered by parsing a malicious web page or a PDF document could perform the same actions to breach the sandbox.

A Better Approach: Hardware Isolated Micro-virtualization
Bromium Inc. has introduced a hardware-enforced protection model called microvirtualization that offers several advantages over traditional sandboxes for security isolation. Researchers in the security industry will recognize the core elements of the Bromium architecture, pictured below, which implements a Least Privilege Separation Kernel (LPSK) using a small hypervisor – termed a microvisor. The microvisor works in concert with special virtualization circuitry in the CPU (VTx on Intel) which enforces the boundaries between the primary trusted and virtual untrusted environments.

The microvisor extends the isolation and protection of hardware virtualization into Windows and its applications, adding a new hardware-protected execution mode for untrustworthy tasks. These micro-VMs are automatically created in a fraction of a second to isolate any task that processes untrusted data or interpreted code, or that accesses an untrusted network.

Windows sees micro-VMs as tasks under its control – it schedules them and tracks their performance and resource use. Micro-VMs are small because they contain only task-specific state, and they run natively with full performance. They offer complete compatibility for all applications, and provide hardware-enforced protection for the desktop, enterprise data, applications and networks. Best of all, micro-VMs do not negatively impact the user experience.
Using this approach is a better solution to the problem because the architecture assumes that the traditional user-space / kernel protection boundary (which the sandboxes studied in this paper attempt to protect) will be compromised, and that entire user mode and kernel environment of an untrusted task may be running arbitrary attacker code.

All resources are accessible in a logically separate view (of the same system), but to access any resource requires an attacker to make a hyper-call, traverse the mandatory access controls enforced by the Microvisor, and then further attack the high privileged system from user mode.

While appealing, this approach is not without risk. The hypervisor and the supporting environment could introduce a new attack vector. The Bromium Microvisor relies on a dramatically reduced code size and its narrow hyper-call interface to dramatically improve resilience to attack. Using this approach, the amount of functionality exposed by a hardened hypervisor to the attacker, although not negligible, is many orders of magnitude smaller than the equivalent Windows OS code – and consequently it is much less vulnerable to attack.

Summary and Conclusions
We have studied two classes of sandboxes

› The “OS Enhancement” or generic sandbox favored by vendors whose goal is to protect the kernel from any of a range of (pre-configured and pre-installed) applications, and

› The “Master / Slave” application-specific sandbox favored by individual application vendors whose applications are vulnerable to exploit by untrusted content or programs

We have shown, through architectural analysis and practical validation that traditional sandboxes:

› Independent of the design or implementation quality of the sandbox in question can be compromised using either design or coding flaws in the underlying operating system it aims to protect and is a part of.

› Microsoft® Windows has many as-yet unidentified (and therefore unpatched) kernel flaws owing to its size and complexity

› OS vulnerabilities in kernel code or interfaces that are accessible even to de-privileged, untrusted sandbox code can be straightforwardly exploited to gain full control over the system.

› It is therefore reasonable to expect a growth in the use of this vector on the part of attackers wishing to escape sandboxed applications. Indeed there are early indications that this is indeed the approach being followed in recent attacks.
We conclude that while the use of sandboxing can help to protect Windows from vulnerable applications, we expect a rapid increase in attacks that successfully bypass sandboxes. Though sandboxing can certainly help, it is at best another Band-Aid in a game of increasingly sophisticated attacks and that a different approach is required to solve the problem of advanced attacks.

We have presented a new class of isolation solution Bromium Microvirtualization that offers a substantially more resilient barrier to protect vulnerable systems.

Bibliography
2) Sandboxie, http://www.sandboxie.com
4) BufferZone Pro, http://www.trustware.com/BufferZone-Pro/
5) http://dev.chromium.org/developers/design-documents/sandbox
7) http://blog.chromium.org/2012/10/pwnium-2-results-and-wrap-up_10.html
8) “Pwn2Own 2012: Google Chrome browser sandbox first to fall” http://www.zdnet.com/blog/security/pwn2own-2012-google-chrome-browser-sandbox-first-to-fall/10588
9) http://blog.chromium.org/2012/11/a-safer-playground-for-your-linux-and.html