Dismantling Smart App Control

By Joe Desimone

Published: 2024-08-06 · Archived: 2026-04-05 19:53:58 UTC

Introduction

Reputation-based protections like Elastic's reputation service can significantly improve detection capabilities while maintaining low false positive rates. However, like any protection capability, weaknesses exist and bypasses are possible. Understanding these weaknesses allows defenders to focus their detection engineering on key coverage gaps. This article will explore Windows Smart App Control and SmartScreen as a case study for researching bypasses to reputation-based systems, then demonstrate detections to cover those weaknesses.

Key Takeaways:

- Windows Smart App Control and SmartScreen have several design weaknesses that allow attackers to gain initial access with no security warnings or popups.
- A bug in the handling of LNK files can also bypass these security controls
- Defenders should understand the limitations of these OS features and implement detections in their security stack to compensate

SmartScreen/SAC Background

Microsoft SmartScreen has been a built-in OS feature since Windows 8. It operates on files that have the "Mark of the Web" (MotW) and are clicked on by users. Microsoft introduced Smart App Control (SAC) with the release of Windows 11. SAC is, in some ways, an evolution of SmartScreen. Microsoft says it "adds significant protection from new and emerging threats by blocking apps that are malicious or untrusted." It works by querying a Microsoft cloud service when applications are executed. If they are known to be safe, they are allowed to execute; however, if they are unknown, they will only be executed if they have a valid code signing signature. When SAC is enabled, it replaces and disables Defender SmartScreen.

Microsoft exposes undocumented APIs for querying the trust level of files for SmartScreen and Smart App Control. To help with this research, we developed a utility that will display the trust of a file. The source code for this utility is available here.

Signed Malware

One way to bypass Smart App Control is to simply sign malware with a code-signing certificate. Even before SAC, there has been a trend towards attackers signing their malware to evade detection. More recently, attackers have routinely obtained Extend Validation (EV) signing certificates. EV certs require proof of identity to gain access and can only exist on specially designed hardware tokens, making them difficult to steal. However, attackers have found ways to impersonate businesses and purchase these certificates. The threat group behind

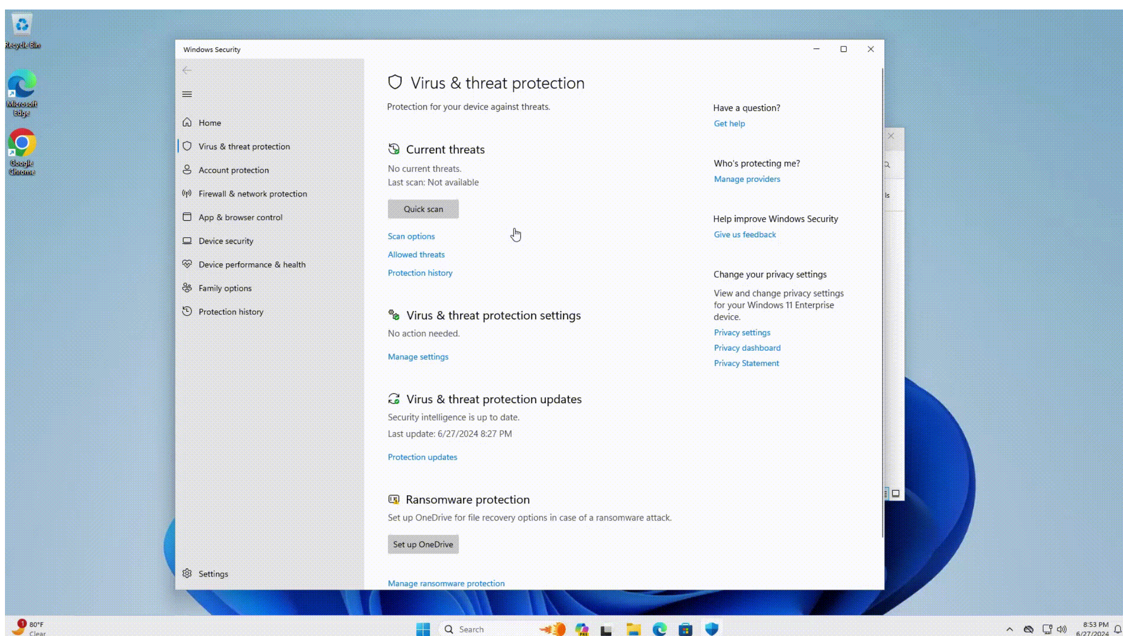
SolarMarker has leveraged over 100 unique signing certificates across their campaigns. Certificate Authorities (CAs) should do more to crack down on abuse and minimize fraudulently-acquired certificates. More public research may be necessary to apply pressure on the CAs who are most often selling fraudulent certificates.

Reputation Hijacking

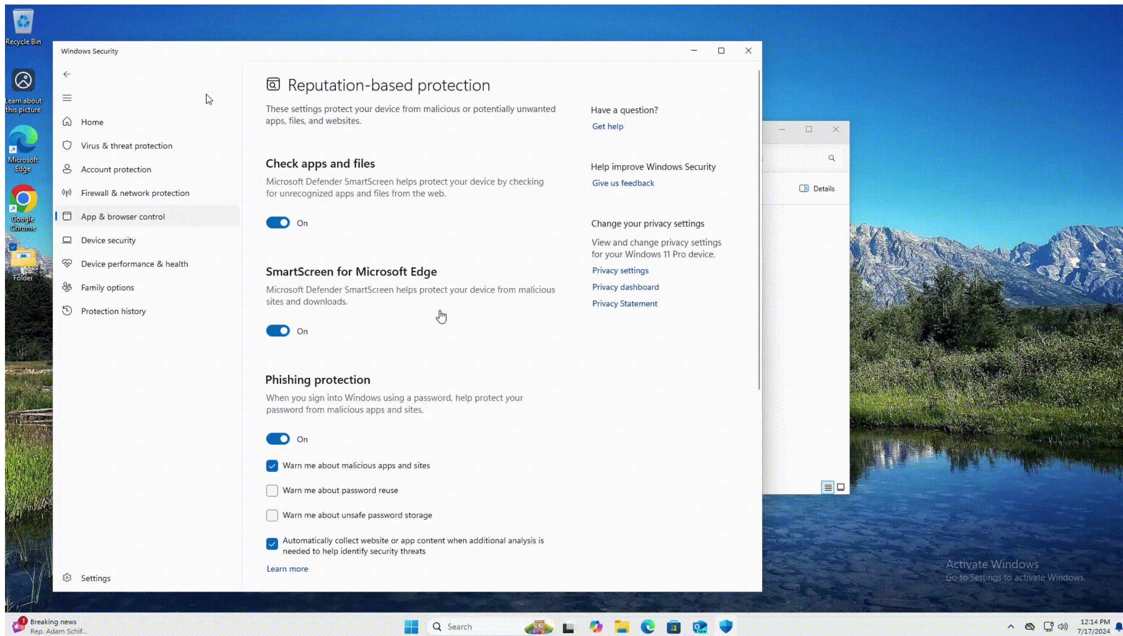
Reputation hijacking is a generic attack paradigm on reputation-based malware protection systems. It is analogous to the misplaced trust research by Casey Smith and others against application control systems, as well as the vulnerable driver research from Gabriel Landau and I. Unfortunately, the attack surface in this case is even larger. Reputation hijacking involves finding and repurposing apps with a good reputation to bypass the system. To work as an initial access vector, one constraint is that the application must be controlled without any command line parameters—for example, a script host that loads and executes a script at a predictable file path.

Script hosts are an ideal target for a reputation hijacking attack. This is especially true if they include a foreign function interface (FFI) capability. With FFI, attackers can easily load and execute arbitrary code and malware in memory. Through searches in VirusTotal and GitHub, we identified many script hosts that have a known good reputation and can be co-opted for full code execution. This includes Lua, Node.js, and AutoHotkey interpreters. A sample to demonstrate this technique is available here.

The following video demonstrates hijacking with the JamPlus build utility to bypass Smart App Control with no security warnings:



In another example, SmartScreen security warnings were bypassed by using a known AutoHotkey interpreter:

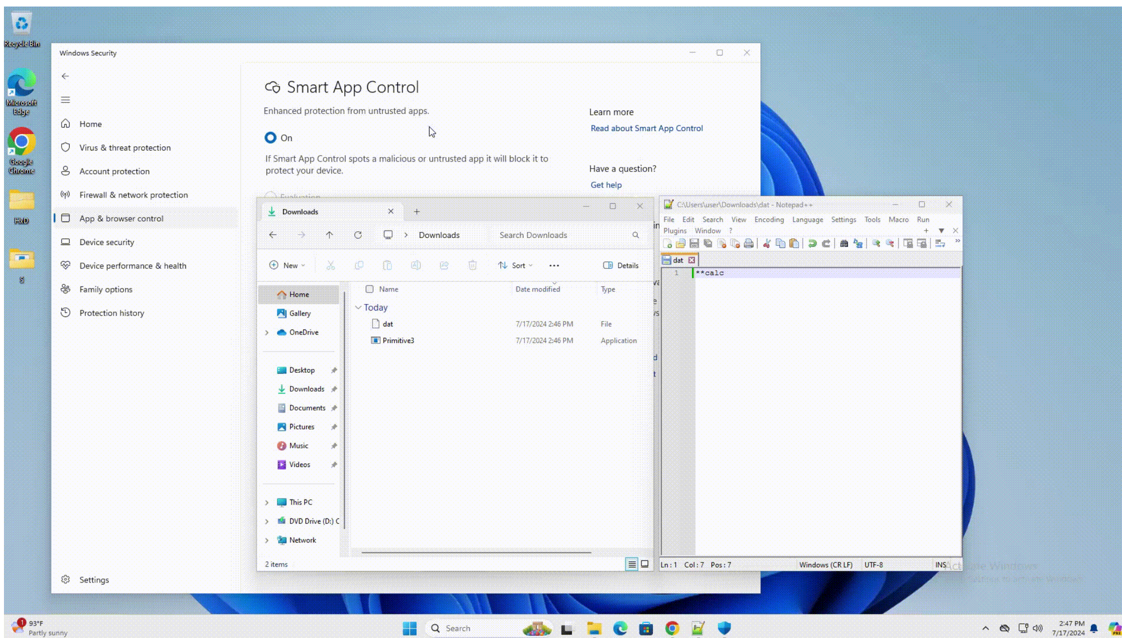


Another avenue to hijack the reputation of a known application is to exploit it. This could be simple, like a classic buffer overflow from reading an INI file in a predictable path. It could be something more complex that chains off other primitives (like command execution/registry write/etc). Also, multiple known apps can be chained together to achieve full code execution. For example, one application that reads a configuration file and executes a command line parameter can then be used to launch another known application that requires a set of parameters to gain arbitrary code execution.

Reputation Seeding

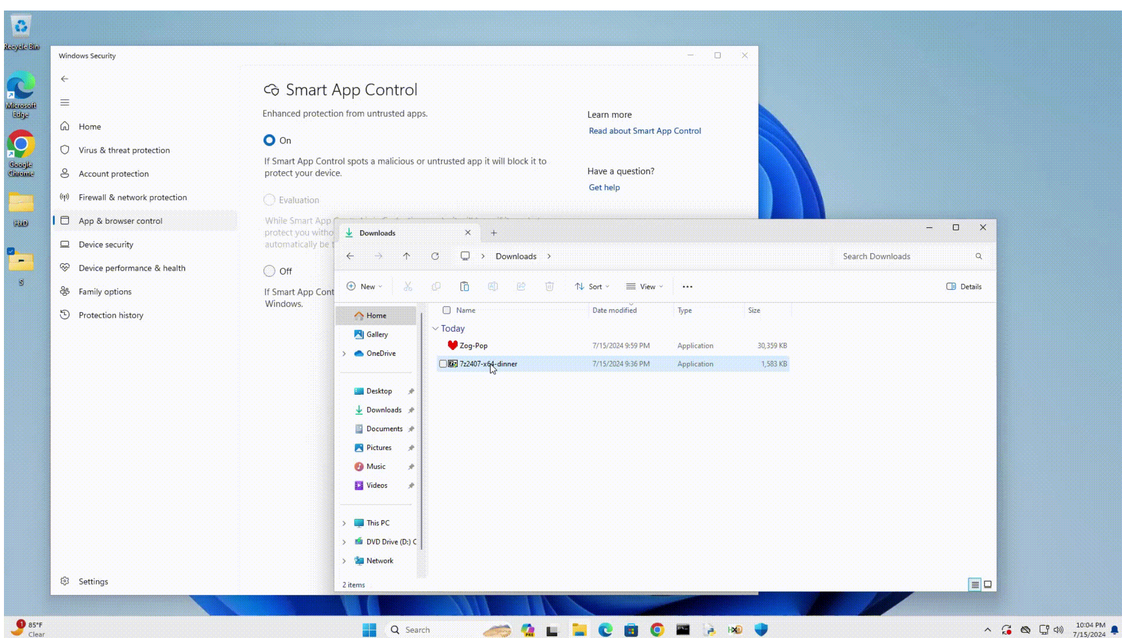
Another attack on reputation protections is to seed attacker-controlled binaries into the system. If crafted carefully, these binaries can appear benign and achieve a good reputation while still being useful to attackers later. It could simply be a new script host binary, an application with a known vulnerability, or an application that has a useful primitive. On the other hand, it could be a binary that contains embedded malicious code but only activates after a certain date or environmental trigger.

Smart App Control appears vulnerable to seeding. After executing a sample on one machine, it received a good label after approximately 2 hours. We noted that basic anti-emulation techniques seemed to be a factor in receiving a benign verdict or reputation. Fortunately, SmartScreen appears to have a higher global prevalence bar before trusting an application. A sample that demonstrates this technique is available [here]() and is demonstrated below:



Reputation Tampering

A third attack class against reputation systems is reputation tampering. Normally, reputation systems use cryptographically secure hashing systems to make tampering infeasible. However, we noticed that certain modifications to a file did not seem to change the reputation for SAC. SAC may use fuzzy hashing or feature-based similarity comparisons in lieu of or in addition to standard file hashing. It may also leverage an ML model in the cloud to allow files that have a highly benign score (such as being very similar to known good). Surprisingly, some code sections could be modified without losing their associated reputation. Through trial and error, we could identify segments that could be safely tampered with and keep the same reputation. We crafted one tampered binary with a unique hash that had never been seen by Microsoft or SAC. This embedded an “execute calc” shellcode and could be executed with SAC in enforcement mode:



LNK Stomping

When a user downloads a file, the browser will create an associated "Zone.Identifier" file in an alternate data stream known as the Mark of the Web (MotW). This lets other software (including AV and EDR) on the system know that the file is more risky. SmartScreen only scans files with the Mark of the Web. SAC completely blocks certain file types if they have it. This makes MotW bypasses an interesting research target, as it can usually lead to bypassing these security systems. Financially motivated threat groups have discovered and leveraged multiple vulnerabilities to bypass MotW checks. These techniques involved appending crafted and invalid code signing signatures to javascript or MSI files.

During our research, we stumbled upon another MotW bypass that is trivial to exploit. It involves crafting LNK files that have non-standard target paths or internal structures. When clicked, these LNK files are modified by explorer.exe with the canonical formatting. This modification leads to removal of the MotW label before security checks are performed. The function that overwrites the LNK files is **_SaveAsLink()** as shown in the following call stack:

```
kernelbase.WriteFile
shcore.Ordinal#100+6BF
shcore.SetCurrentProcessExplicitAppUserModelID+8A4
windows.storage.CShellLink::_SaveAsLink+9B
windows.storage.CShellLink::_SaveToFile+112
windows.storage.CShellLink::Save+B4
windows.storage.public: virtual long __cdecl CLibrariesFolderB
windows.storage.CShellLink::InvokeCommand+58
shell32.public: virtual long __cdecl CDefFolderMenu::InvokeCom
shell32.long __cdecl SHInvokeCommandOnContextMenu2(struct HWND_
shell32.unsigned long __cdecl s_DoInvokeVerb(void *)+D3
shcore.SHReleaseThreadRef+1CDD
kernel32.BaseThreadInitThunk+1D
ntdll.RtlUserThreadStart+28
```

The function that performs the security check is **CheckSmartScreen()** as shown in the following call stack:

```
windows.storage.CheckSmartScreenWithAltFile
windows.storage.CBindAndInvokeStaticVerb::CheckSmartScreen+100
windows.storage.CBindAndInvokeStaticVerb::Execute+16A
windows.storage.RegDataDrivenCommand::_TryInvokeAssociation+C1
windows.storage.RegDataDrivenCommand::_Invoke+134
shell32.CRegistryVerbsContextMenu::_Execute+10F
shell32.CRegistryVerbsContextMenu::InvokeCommand+E9
shell32.public: virtual long __cdecl CDefFolderMenu::InvokeCommand(struct _C
windows.storage.CShellLink::_InvokeDirect+16B
windows.storage.CShellLink::_ResolveAndInvoke+F6
windows.storage.CShellLink::InvokeCommand+58
shell32.public: virtual long __cdecl CDefFolderMenu::InvokeCommand(struct _C
shell32.long __cdecl SHInvokeCommandOnContextMenu2(struct HWND__*, struct IU
shell32.unsigned long __cdecl s_DoInvokeVerb(void *)+D3
shcore.SHReleaseThreadRef+1CDD
kernel32.BaseThreadInitThunk+1D
ntdll.RtlUserThreadStart+28
```

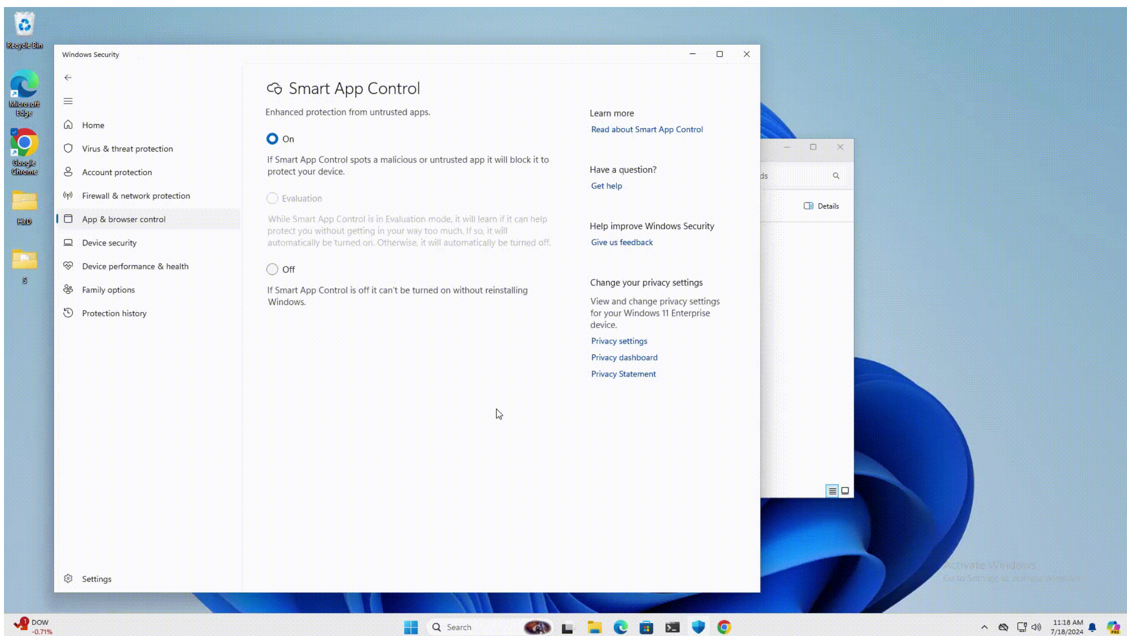
The easiest demonstration of this issue is to append a dot or space to the target executable path (e.g., `powershell.exe.`). Alternatively, one can create an LNK file that contains a relative path such as `.\target.exe` . After clicking the link, `explorer.exe` will search for and find the matching `.exe` name, automatically correct the full path, update the file on disk (removing MotW), and finally launch the target. Yet another variant involves crafting a multi-level path in a single entry of the LNK's target path array. The target path array should normally have 1 entry per directory. The pylnk3 utility shows the structure of an exploit LNK (non-canonical format) before and after execution (canonical format):

```
<LinkTargetIDList>:
  <RootEntry: MY_COMPUTER>
  <DriveEntry: b'C:'>
  <PathSegmentEntry: windows\System32\WindowsPowerShell\v1.0\powershell.exe>
```

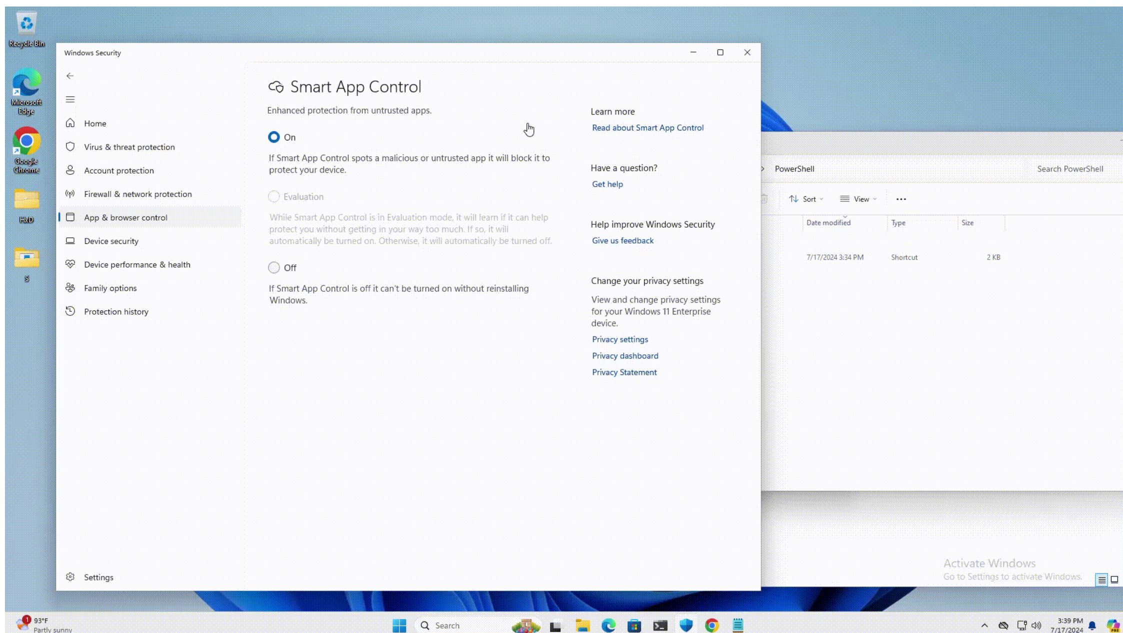
```
<LinkTargetIDList>:
  <RootEntry: MY_COMPUTER>
  <DriveEntry: b'C:'>
  <PathSegmentEntry: Windows>
  <PathSegmentEntry: System32>
  <PathSegmentEntry: WindowsPowerShell>
  <PathSegmentEntry: v1.0>
  <PathSegmentEntry: powershell.exe>
```

A Python script that demonstrates these techniques is available [here](#).

The following shows an LNK file bypassing MotW restrictions under Smart App Control to launch Powershell and pop calc:



In another example, we show this technique chained with the Microsoft cdb command line debugger to achieve arbitrary code execution and execute shellcode to pop calc:



We identified multiple samples in VirusTotal that exhibit the bug, demonstrating existing in the wild usage. The oldest sample identified was submitted over 6 years ago. We also disclosed details of the bug to the MSRC. It may be fixed in a future Windows update. We are releasing this information, along with detection logic and countermeasures, to help defenders identify this activity until a patch is available.

Detections

Reputation hijacking, by its nature, can be difficult to detect. Countless applications can be co-opted to carry out the technique. Cataloging and blocking applications known to be abused is an initial (and continual) step.

```
 process where process.parent.name == "explorer.exe" and process.hash.sha256 in (
"ba35b8b4346b79b8bb4f97360025cb6befaf501b03149a3b5fef8f07bdf265c7", // AutoHotKey
"4e213bd0a127f1bb24c4c0d971c2727097b04eed9c6e62a57110d168ccc3ba10" // JamPlus
)
```

However, this approach will always lag behind attackers. A slightly more robust approach is to develop behavioral signatures to identify general categories of abused software. For example, we can look for common Lua or Node.js function names or modules in suspicious call stacks:

```
 sequence by process.entity_id with maxspan=1m
[library where
  (dll.Ext.relative_file_creation_time <= 3600 or
   dll.Ext.relative_file_name_modify_time <= 3600 or
   (dll.Ext.device.product_id : ("Virtual DVD-ROM", "Virtual Disk","USB *") and not dll.path : "C:\\*")) and
   _arraysearch(process.thread.Ext.call_stack, $entry, $entry.symbol_info: "*!luaopen_*")] by dll.hash.sha256
[api where
 process.Ext.api.behaviors : ("shellcode", "allocate_shellcode", "execute_shellcode", "unbacked_rwx", "rwx", "h
 process.thread.Ext.call_stack_final_user_module.hash.sha256 : "?*"] by process.thread.Ext.call_stack_final_use
```

```
api where process.Ext.api.name : ("VirtualProtect*", "WriteProcessMemory", "VirtualAlloc*", "MapViewOfFile*") a
process.Ext.api.behaviors : ("shellcode", "allocate_shellcode", "execute_shellcode", "unbacked_rwx", "rwx", "ho
process.thread.Ext.call_stack_final_user_module.name : "ffi_bindings.node"
```

Security teams should pay particular attention to downloaded files. They can use local reputation to identify outliers in their environment for closer inspection.

```
 from logs-* |
where host.os.type == "windows"
and event.category == "process" and event.action == "start"
and process.parent.name == "explorer.exe"
and (process.executable like "*Downloads*" or process.executable like "*Temp*")
and process.hash.sha256 is not null
| eval process.name = replace(process.name, " \\(1\\).", ".")
| stats hosts = count_distinct(agent.id) by process.name, process.hash.sha256
| where hosts == 1
```

LNK stomping may have many variants, making signature-based detection on LNK files difficult. However, they should all trigger a similar behavioral signal- `explorer.exe` overwriting an LNK file. This is especially anomalous in the downloads folder or when the LNK has the Mark of the Web.

```
file where event.action == "overwrite" and file.extension : "lnk" and
process.name : "explorer.exe" and process.thread.Ext.call_stack_summary : "ntdll.dll|*|windows.storage.dll|she
(
 file.path : ("?:\\Users\\*\\Downloads\\*.lnk", "?:\\Users\\*\\AppData\\Local\\Temp\\*.lnk") or
 file.Ext.windows.zone_identifier == 3
 )
```

Finally, robust behavioral coverage around common attacker techniques such as in-memory evasion, persistence, credential access, enumeration, and lateral movement helps detect realistic intrusions, including from reputation hijacking.

Conclusion

Reputation-based protection systems are a powerful layer for blocking commodity malware. However, like any protection technique, they have weaknesses that can be bypassed with some care. Smart App Control and SmartScreen have a number of fundamental design weaknesses that can allow for initial access with no security warnings and minimal user interaction. Security teams should scrutinize downloads carefully in their detection stack and not rely solely on OS-native security features for protection in this area.

Source: https://www.elastic.co/security-labs/dismantling-smart-app-control