Crypting

The Art of Detection Evasion

Ethan Linton

CISSP, CCNP, MCS 1st Class H.

Educational and Research purposes only

Preface

This research paper examines antivirus-evasion techniques with the singular aim of advancing defensive cyber-security practice and academic discussion. All examples and experimental results were produced and tested in a strictly controlled environment on systems for which the author had full legal authority. Minimal to no code fragments have been included in this publication to reiterate the author's disapproval of any malicious application of the information contained in this paper. Readers are reminded that, across Australia, gaining unauthorised access to, modifying, or impairing computer data or communications is a criminal offence under both State and Commonwealth legislation.

Part 6 of the Crimes Act 1900 prohibits (among other conduct):

- *Unauthorised access to or modification of restricted data (s 308H)*
- *Unauthorised modification of data with intent to cause impairment (s 308D)*
- *Unauthorised impairment of electronic communications (s 308E)*

Each carrying penalties of up to 10 years' imprisonment depending on the circumstances.

At the federal level, Division 477 of the Criminal Code Act 1995 (Cth) criminalises unauthorised access, modification or impairment executed with intent to commit a serious offence (s 477.1), while Division 478 addresses unauthorised access to restricted data and other computer-related harms (s 478.1–478.2) These provisions operate alongside the Cybercrime Act 2001 (Cth), which extends jurisdiction and investigative powers for serious computer offences.

The author emphatically disavows any malicious application of the information contained in this paper. The techniques described must only be used with the explicit, informed consent of the system owner and in compliance with all relevant laws, ethical guidelines, and organisational policies. Nothing in this paper should be construed as legal advice; practitioners should seek professional counsel before conducting security testing outside a controlled setting.

By proceeding beyond this preface, the reader affirms their understanding that the knowledge presented is intended exclusively for lawful research, defensive security work, and the collective strengthening of cyber-resilience.

Table of Contents

Executive Summary

This project, titled "Crypting: The Art of Detection Evasion," investigates advanced techniques utilised to evade anti-virus (AV) and endpoint detection and response (EDR) solutions. The paper begins by defining crypting and categorising different crypters, including static, polymorphic, and metamorphic types, alongside their associated evasion strategies.

Practical examples illustrate comprehensive evasion implementations such as module stomping designed to bypass both static and dynamic detection systems. Specific attention is given to this implementation, where legitimate modules in memory have regions in which are overwritten with custom code to evade security measures.

The author conducted experiments demonstrating the techniques effectiveness, notably achieving evasion from a majority of anti-virus solutions, including real-time defences included within Windows Defender and Malwarebytes. Various supplementary tactics, such as encrypted payloads with runtime decryption, direct syscall invocations, and reflective PE loading, further highlight the complexity and precision of evasion methods.

The research emphasises the importance of defensive cybersecurity awareness and detailed technical understanding, urging security practitioners to adopt proactive threat hunting and robust defence-in-depth strategies. Ethical and legal considerations underpin this paper, reinforcing the necessity of explicit consent and adherence to relevant laws when employing such techniques for educational or defensive purposes.

Crypting

My life for the past four years has had a large focus on the world of cryptography, evidentially through my occupation but also through genuine interest. I always knew about the world of crypting and was interested in exploring it more – what you're reading is the outcome of my exploration.

So, what is crypting? In layman's terms, it's a way of hiding code from detection – in other words:

- It utilises symmetric encryption algorithms such as AES to encrypt what is called the "payload."
- Creates a delivery application to decrypt the above-mentioned encrypted payload and executes this payload
    - This delivery application is obfuscated using techniques such as garbage code insertion, register renaming, and control flow flattening.
- The delivery application and payload are then packed together, thus outputting a single file that (once run) will call upon the delivery application to decrypt and execute the payload via reflective loading, process hallowing, etc.

This all seems straightforward – so why do we have crypting services that charge $20 per file, $100 per crypt, or even $6,000 monthly? They each hold bypass claims with somewhat of a reputation to back it up. These bypass claims include anti-memory dump, and SmartScreen bypasses.

Crypting is an extensive term in relation to anti-virus evasion. There are many different types of crypters, evasion techniques, and methodologies – and I'll be covering a few throughout this post, including demonstrating a practical example that utilises module stomping alongside many more techniques (more on that later) to achieve the bypass of static and runtime analyses across *many* anti-virus providers.

Types of Crypters

Depending on how encryption is applied to payloads, there are different types of crypters, such as Static, Polymorphic, and Metamorphic.

Static crypters use fixed encryption and decryption routines, meaning they use the same encryption key each time. This makes static crypters easily detectable via static analysis (signature detection).

Polymorphic crypters use dynamic encryption and decryption routines – this modification of the encryption algorithm allows for the generation of a unique version

for each delivery application and payload. This modification ensures that static-based analysis detection is more challenging.

Metamorphic crypters combine the encryption elements of polymorphic crypters with completely rewritten delivery application and code at each generation.

Included in the types of crypters are the evasion techniques it executes to avoid scan time and runtime detections.

Scan time crypters are designed to avoid detection while it's being scanned by anti-virus software – this typically means when the file is saved to disk, downloaded or transferred. Anti-virus software uses signatures and heuristics to detect – this means if the payload is encrypted and the delivery application is obfuscated, the anti-virus software sees it as a clean file. However, once the file is run, behaviour-based anti-virus kicks in – this is why we chain scan time and runtime evasion techniques together.

Runtime crypters avoid detection while the code is running on the host system. Techniques utilised by runtime crypters include reflective DLL injection, process hollowing, PE injection, module stomping and API obfuscation. Multiple strategies are employed at once to avoid runtime detection – it's unlikely that a runtime crypter will use just one technique.

Detection Evasion Techniques

This is where the fun begins… I will not be covering every technique, nor will I be going too in-depth – I will primarily provide information surrounding only a few detection evasion techniques.

Module Stomping

This is the fundamental technique that makes up my detection evasion strategy. So… what is module stomping?

Picture an art heist. The thieves don’t conduct a loud break in - they stroll in wearing the museum's uniforms, swap the genuine painting with a perfect-looking fake, and walk out while security still thinks everything is normal.
That, in essence, is what module stomping does inside a running process.

When Windows loads a legitimate DLL, the operating system grants it a trusted place in memory - much like a museum display. Instead of injecting a foreign payload, it quietly overwrites the code inside that already-trusted DLL. The outside frame - the file on disk, the export table, the digital signature - remains untouched, so anti-virus and EDR

sensors keep "seeing" the original work. Meanwhile, the code now hanging behind the frame carries out its own agenda while appearing as the original. By essentially hijacking what's already loaded – it's probable to dodge the most tell-tale signs of interference.

Why use Module Stomping?

Because anti-virus engines usually trust DLLs that originate from reputable locations, tampering with one of those already-loaded libraries allows the payload to slip seamlessly into the host process. Since the DLL you have on disk is essentially untouched and the payload has been executed entirely in memory – static detection evasion occurs.

Module Stomping Process

1. Select a trusted module:
    a. Identify a legitimate DLL already loaded in memory – this would often be a standard Windows module signed and trusted by anti-virus software. Think of ntdll.dll or kernel32.dll.
2. Memory overwrite
    a. Reserve a memory region within the address space of the selected trusted module, ensuring the allocated region has both write and execute permissions. Overwrite this reserved memory region via shellcode injection. At this stage, the original legitimate DLL on disk remains untouched – thus *typically* avoiding detection by anti-virus.
3. Maintain the façade
    a. Keep the DLL operational by tweaking only a narrow portion- say, patching one routine or rerouting specific calls - so the module still behaves normally and avoids detection.

Module Stomping Example

I experimented with creating a crypter using C++ (it's the easiest for this, I tried Go, and that was just a mess) designed to transform typically detectable shellcode into payloads that evaded detection by 68 out of 72 anti-virus products tested on VirusTotal.com, including bypassing real-time protections provided by Windows Defender and Malwarebytes.

How was this achieved?

An endless list of evasion techniques could potentially be applied to evade detection. The following are only a few that I found interesting. Please note that maliciously implementing the below evasion techniques **will** get you detected.

1. **Encrypted payload + Runtime Decryption**
    a. The shellcode that lives on disk is XOR-encrypted, so anti-virus scanners never see its raw bytes. **Note:** I know… XOR encryption?! Please don't use

XOR encryption for actual encryption use cases... this is just so static analysis, aka signature detection, doesn't pop off since the shellcode is scrambled.

   b. At runtime, a tiny multi-byte XOR loop reconstructs the real code in memory just before execution.

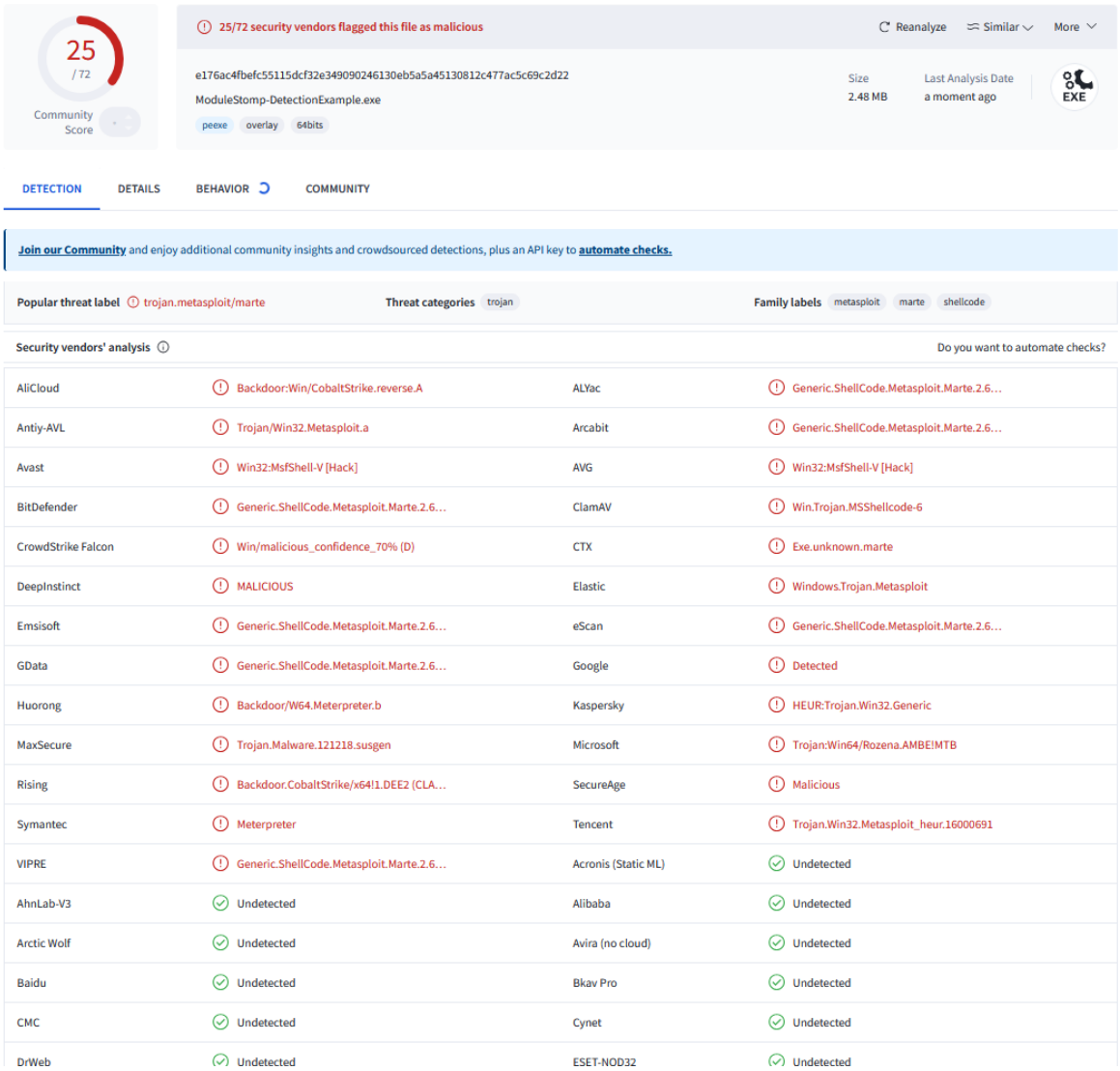
2. **Junk Operations to Foil Emulators**
   a. This prevents the compiler from optimising the loop away and adds "noise" that can confuse sandbox detectors.
3. **String-Free API Lookup via Export-Table Hashing**
   a. For example, rather than embedding the plain ASCII names "NtAllocateVirtualMemory" or calling GetProcAddress, it stores only 32-bit DJB2 hashes (e.g. 0x6793C34C) and walks ntdll.dll's export directory at runtime to find the matching export.
   b. This removes any tell-tale "VirtualAlloc" or "CreateThread" strings from the binary, thus tricking scanners that look for suspicious imports. When it comes to detection evasion, anything counts.
4. **Direct Syscall Invocation**
   a. This means directly invoking NtAllocateVirtualMemory, NtProtectVirtualMemory, and NtCreateThreadEx, bypassing the typical kernel32.dll wrappers.
   b. That means any AV / EDR user-mode hooks on VirtualAlloc, VirtualProtect or CreateThread never get called, and you land straight in the kernel syscall.
5. **RWX Memory without VirtualAlloc**
   a. Using the NtAllocateVirtualMemory syscall it's possible to carve out a read-write-execute page instead of relying on the higher-level (and more commonly hooked) VirtualAlloc.
6. **In-Memory Thread Creation**
   a. Rather than CreateRemoteThread or RtlCreateUserThread, using NtCreateThreadEx to spin up a thread directly against its hidden, freshly allocated region could potentially meet this requirement. Through in-memory thread creation, it is probable to side-step common user-mode patch points.
7. **Minimal C Runtime & Imports**
   a. This means keeping your IAT footprint lean and clean.
8. **Clean-Up**
   a. After execution, restore page protections and free the RWX region to erase any artifacts of the decrypted shellcode.

In other words... a typical evasion loader first decrypts its embedded, obfuscated payload entirely in memory. It then loads a fresh copy of a harmless system DLL without

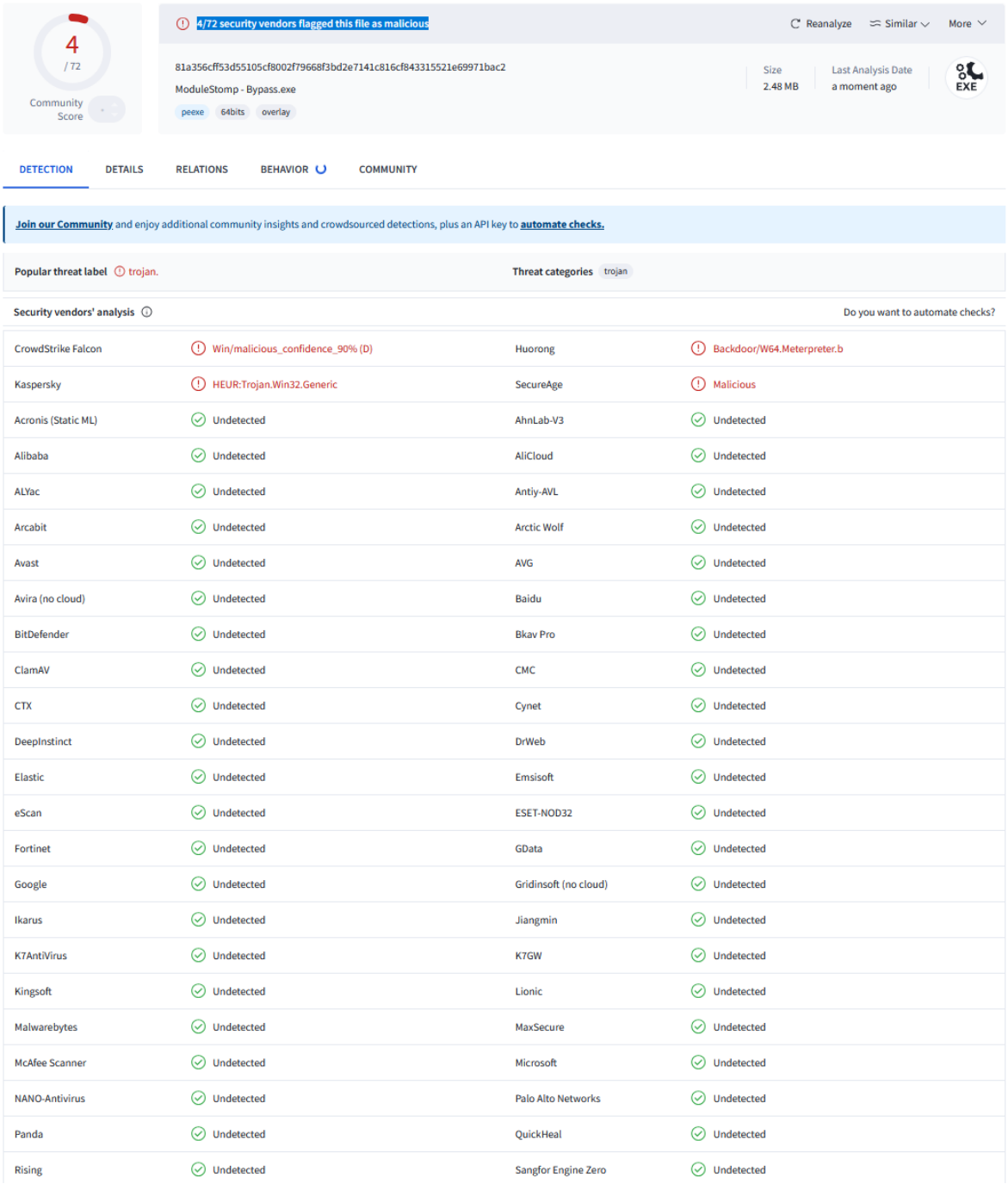
running its initialisation routines, parses the DLL's in-memory headers to find the start of its code section (.text), and then makes that section writable and executable. At the DLL's entry point, it overwrites the original bytes with the decrypted payload, flushes the CPU instruction cache, and creates a new thread beginning at that overwritten entry point. In effect, the legitimate DLL has been "stomped" in place and now silently runs the injected code instead of its own, evading on-disk scanning and hiding under the guise of a trusted module.

Here are a few screenshots throughout the journey of evading detection.

The first screenshot below shows a scan conducted via Virustotal on the shellcode within the compiled .exe file. As you can see, I am using shellcode, which is generally detected when in raw format.



The following screenshot displays another scan conducted by Virustotal after applying a few unspecified evasion techniques. You can see that it's not "FUD" (that's what the cool kids say; it means fully undetectable) – but generally, in a sense, these people who sell this as a service typically tell you NOT to scan your application using VirusTotal.



4 / 72

Community Score

4/72 security vendors flagged this file as malicious

Reanalyze Similar More

81a356cff53d55105cf8002f79668f3bd2e7141c816cf843315521e69971bac2

ModuleStomp - Bypass.exe

peexe 64bits overlay

Size 2.48 MB

Last Analysis Date a moment ago

EXE

DETECTION DETAILS RELATIONS BEHAVIOR COMMUNITY

Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label trojan.

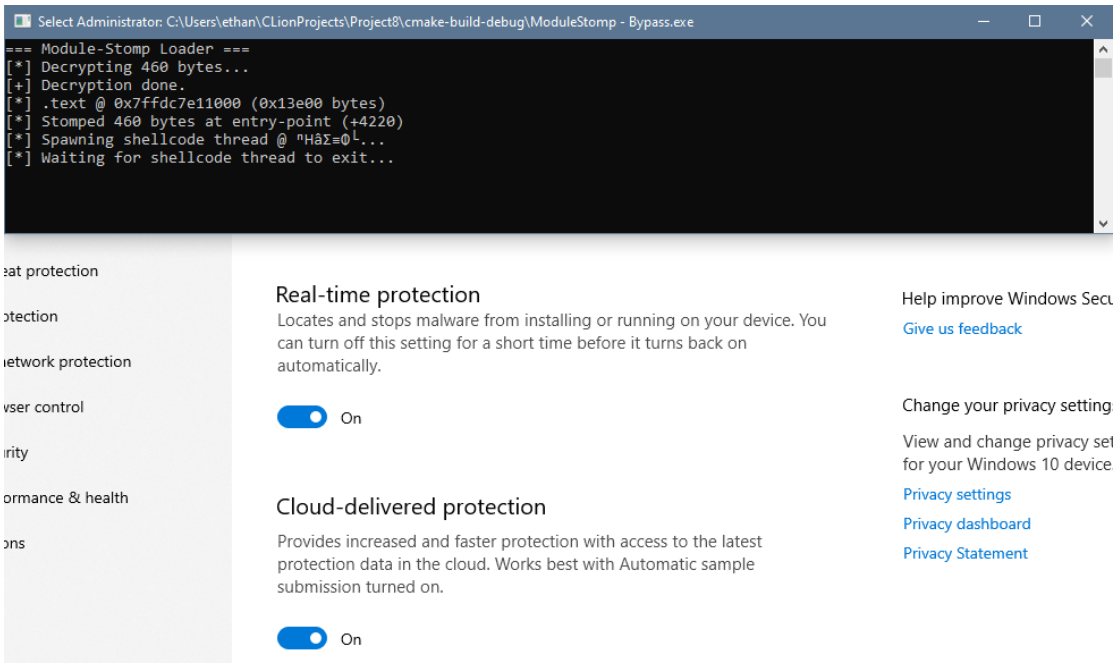
Threat categories trojan

Security vendors' analysis

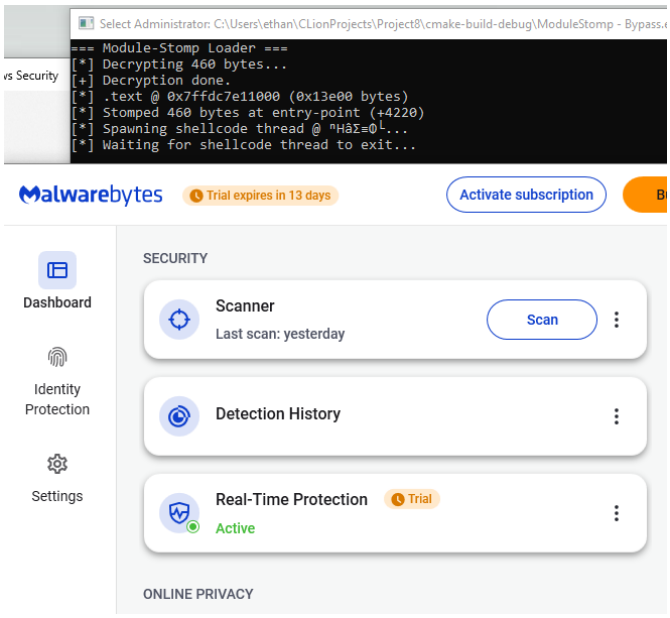
Do you want to automate checks?

CrowdStrike Falcon	Win/malicious_confidence_90% (D)	Huorong	Backdoor/W64.Meterpreter.b
Kaspersky	HEUR:Trojan.Win32.Generic	SecureAge	Malicious
Acronis (Static ML)	Undetected	AhnLab-V3	Undetected
Alibaba	Undetected	AliCloud	Undetected
ALYac	Undetected	Antiy-AVL	Undetected
Arcabit	Undetected	Arctic Wolf	Undetected
Avast	Undetected	AVG	Undetected
Avira (no cloud)	Undetected	Baidu	Undetected
BitDefender	Undetected	Bkav Pro	Undetected
ClamAV	Undetected	CMC	Undetected
CTX	Undetected	Cynet	Undetected
DeepInstinct	Undetected	DrWeb	Undetected
Elastic	Undetected	Emsisoft	Undetected
eScan	Undetected	ESET-NOD32	Undetected
Fortinet	Undetected	GData	Undetected
Google	Undetected	Gridinsoft (no cloud)	Undetected
Ikarus	Undetected	Jiangmin	Undetected
K7AntiVirus	Undetected	K7GW	Undetected
Kingsoft	Undetected	Lionic	Undetected
Malwarebytes	Undetected	MaxSecure	Undetected
McAfee Scanner	Undetected	Microsoft	Undetected
NANO-Antivirus	Undetected	Palo Alto Networks	Undetected
Panda	Undetected	QuickHeal	Undetected
Rising	Undetected	Sangfor Engine Zero	Undetected

So that's all cool... it's essentially evading static detection, but what about runtime detection? The two screenshots below display this capability. The top window is the "ModuleStomp – Bypass.exe" shown in the previous Virustotal screenshot. This executable runs shellcode, which would otherwise be detectable. The command prompt displays a list of very generalised steps for debugging purposes. The window below is Windows Defender running concurrently with real-time protection enabled.

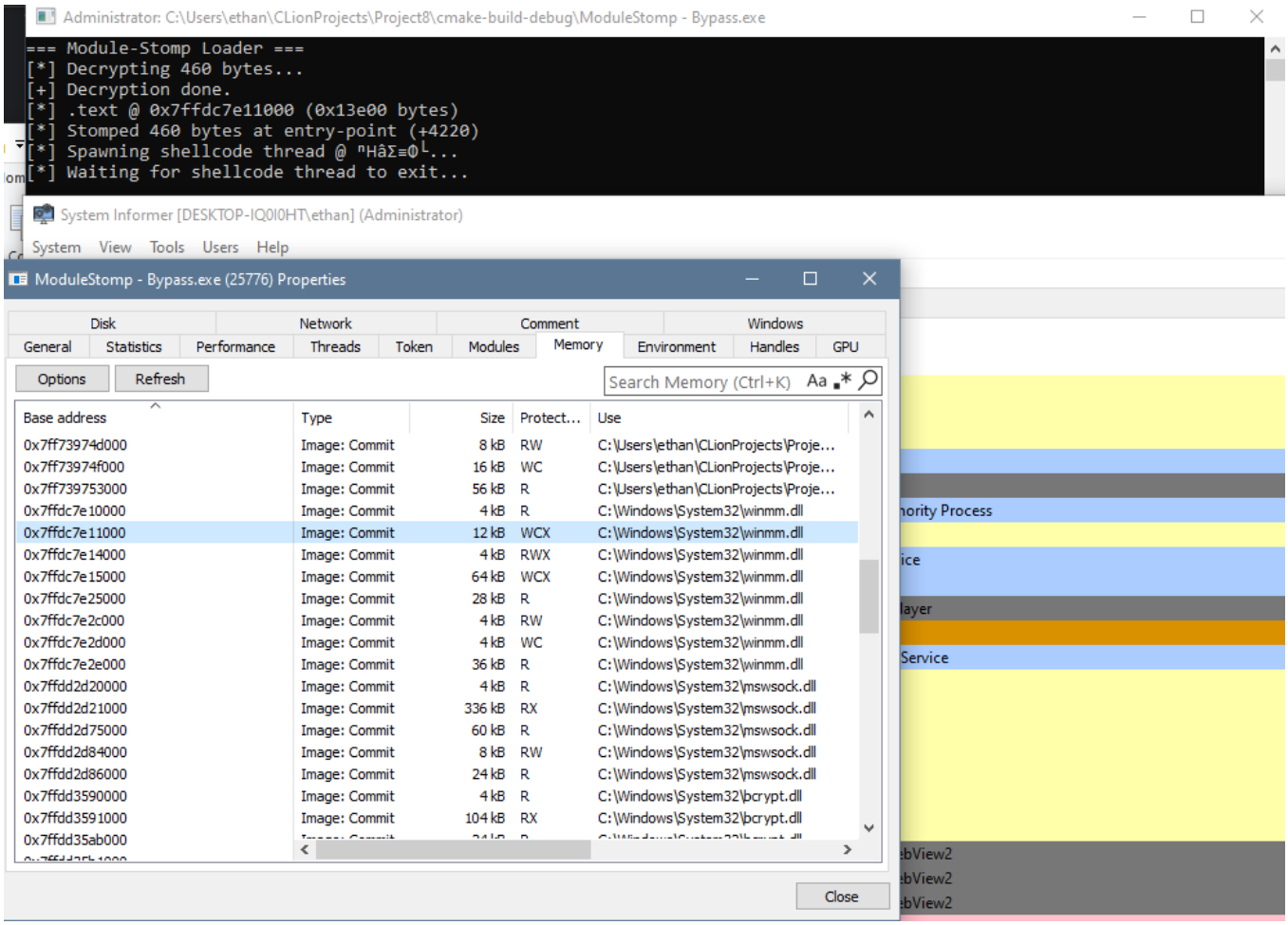


The screenshot below (apart of a collection of two as noted above) displays the same ModuleStomp – Bypass.exe executable file running concurrently alongside Malwarebytes real-time protection. This here is another showcase of the evasion capabilities surrounding runtime detection.



The following screenshot below - is the Memory view of "ModuleStomp – Bypass.exe" process in System Informer. It shows you the virtual address ranges for all the mapped modules, and right in the middle, you can see the pages belonging to C:\Windows\System32\winmm.dll around 0x7ffdc7e10000.

Notice how one of those regions is marked WCX (write-copy-execute) instead of the normal read-only or read-execute you'd see for a DLL's code section. That's the region that the module stomped on when it overwrote the .text section with the decrypted shellcode and then changed its protection so the loader could execute from it. In short, this displays the "module stomping" aspect of the evasion technique.



This essentially concludes the practical section of this paper. Comprehensive details have been left out to combat any malicious application using the details that were written above.

Alternative Data Stream

An often-overlooked NTFS feature is Alternate Data Streams (ADS). Every NTFS file can carry one or more hidden "streams" of data that don't appear in Explorer or a standard dir. Windows itself will happily serve them up if you open the file with the stream name appended (filename:streamname).

By placing your encrypted payload into an ADS attached to your delivery application, you:

1. Never touch the visible filesystem with a raw shellcode blob.
2. Still, ship everything in one executable file.
3. Only read the ADS at runtime, decrypt it into memory, and execute it, leaving no payload lying around on disk.

Process Injection & Hollowing

Instead of overwriting a DLL, the loader carves out a legitimate process ("svchost.exe," for example), unmaps its code section, writes in its own payload, fixes up imports/relocations, and then resumes execution.

Because your code runs under the original PID, with all its loaded modules and privileges, many AVs that watch for standalone processes will see an original binary conducting normal operation - until your injected thread suddenly reaches out to your callback.

Reflective PE Loading

Here, a PE image is carried as data, mapped wholly in memory, and manually linked (headers, sections, base relocations and imports resolved) by the loader - no LoadLibrary calls and no disk drop.

Because this happens out of sight of the OS loader - and because you never create any on-disk artifacts - most static or simple heuristic scanners won't notice. Your reflective loader sits in a small delivery application that itself was injected or allocated earlier, decrypts or unpacks the real PE image, and then "reflectively" loads it into memory. The result is a second stage - or entire payload - that looks like it sprang fully formed from RAM, with no traces on disk and no normal loader calls to flag.

Conclusion

In closing, it's essential to recognise that effective evasion is not the result of any single trick but rather the careful orchestration of multiple techniques – as listed throughout this paper and beyond - all working together to slip past defensive controls. This paper has walked through the foundations of crypting, surveyed the various categories of crypters, presented evasion implementation examples, and examined various anti-analysis and anti-detection strategies. While these methods can be fascinating from a technical standpoint, they also carry significant ethical and legal implications. Any deployment of such capabilities must be conducted under proper authorisation and in strict compliance with applicable laws. By understanding both the power and the responsibility that comes with advanced evasion, security professionals can better defend against it - and researchers can continue to push forward on the right side of the line.

Thank you to Sam Anttila, S12, and INTEL471.