This talk is a defender's perspective on Windows memory scanning – with a particular focus of the protection fluctuation evasion approach introduced by Gargoyle in 2017 and how to find the footprints of such hidden shellcode.

whoami

Security Research Engineer at Elastic

- Elastic Defend (“EDR”) developer
- Elastic Security Labs Blogger
- https://www.elastic.co/blog/author/john-uhlmann

elastic

elastic security labs

Agenda

1. Why do security products scan memory?
2. Memory scanning & evasion recap
3. Detection opportunities for hidden shellcode
   - Detection via immutable code page principle violations
   - Detection via CFG bitmap anomalies
4. Hunting via process behaviour summaries

Three tools released.

Why do security products scan memory?

- On Windows x64, Microsoft has –
- hardened the kernel,
- claimed the hypervisor.

Microsoft used the release of 64-bit Windows to Microsoft made some critical security improvements.
They hardened the kernel with Driver Signing Enforcement and PatchGuard, and now Virtualization-Based Security (VBS).

These have all had a strong positive impact on Windows Security, but…

Why do security products scan memory?

- On Windows x64, Microsoft has –
- hardened the kernel,
- claimed the hypervisor, and
- made private user-mode executable memory an indefensible boundary for kernel-mode security products.

- This just leaves memory scanning.
- It's not perfect, but it's still a valuable defensive layer.

…they have inadvertently made private user-mode executable memory an indefensible boundary.

For each LoadLibrary() call security products receive a kernel callback – an inline opportunity for prevention before that code executes. For each VirtualProtect(RX) security products do not.

A security boundary for private executable memory cannot reasonably be retrofitted by security vendors.

Kernel hooks are not supported - for good reasons.
User-mode hook approaches are common (so arguably any performance concerns are moot) but these are trivally bypassed.
Hypervisor shadow hooks are technically possible - but these are incompatible with VBS.
There is Arbitrary Code Guard and it is theoretically an extremely powerful protection, it prevents any modification of executable code. But its applicability is extremely limited and is not widely deployed in practice.

This just leaves memory scanning. It's not perfect, but it's still a valuable defensive layer.

Overview of memory scanners

Generic Scanners

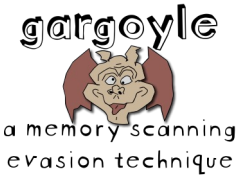
- YARA - memory content signatures
- PE-sieve - image metadata anomalies and content heuristics
- Moneta - memory metadata anomalies

The current state of memory scanning is best understood by the big three open-source scanners.

Evasion – key concept

"a common technique for reducing computational burden is to limit analysis on executable code pages only" - Josh Lospinoso

https://lospi.net/security/assembly/c/cpp/developing/software/2017/03/04/gargoyle-memory-analysis-evasion.html




```
VirtualProtect(pShellcode, sizeof(shellcode), PAGE_READWRITE, &OldProtect);
```


The current state of memory scanning evasion boils down to Gargoyle.
This was truly a seminal evasion approach. Six years later and it is still viable.
Almost all of the current memory scanning evasion approaches rely on this key concept introduced by Josh.
That memory scanning is limited to executable code pages only.

But Josh was half right. Scanning non-executable memory is computationally expensive (there is a lot of it) – but it is also extremely false positive prone.
Security tools, disassemblers, hex editor, browsers open to malware analysis blogs etc.
There are numerous legitimate reasons to have malicious indicators in these benign memory regions.
So scanning all memory is likely to never be an option even with GPU offload and other optimisations.

The crux of the evasion is periodic VirtualProtect calls – and specifically that it can be used remove all references to a region having previously been executable (or concurrently writable and executable) from the Virtual Address Descriptor tree maintained by the kernel memory manager. The VAD tree only stores the original allocation protection and the current protection.

Evasion recap

- Gargoyle - memory protection fluctuation via APC timer and ROP chain
- obfuscate-and-sleep - encrypted state fluctuation via post-sleep stub
- FOLIAGE - encrypted state fluctuation via APC timers and context manipulation
- Shellcode Fluctuation - memory protection fluctuation via post-sleep indirect stub
- DeepSleep - memory protection fluctuation via post-sleep ROP chain
- Ekko - encrypted state fluctuation via timer queues and context manipulation

- Scheduled Tasks ;-)

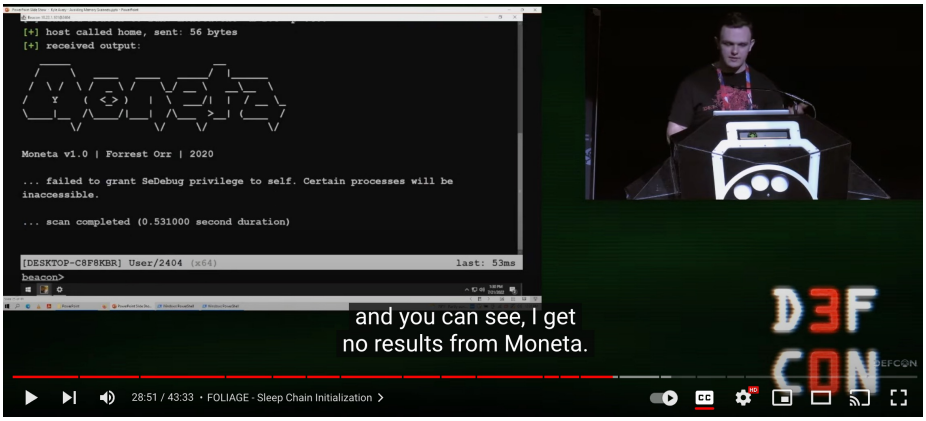
Since Gargoyle, there have been a number of Gargoyle-esque variants.

But quickly we have –

- Gargoyle - memory protection fluctuation via APC timer and ROP chain
- CoblatStrike's obfuscate-and-sleep - encrypted state fluctuation via post-sleep stub
- FOLIAGE - encrypted state fluctuation via APC timers and context manipulation
- Shellcode Fluctuation - memory protection fluctuation via post-sleep indirect stub
- DeepSleep - memory protection fluctuation via post-sleep ROP chain
- Ekko - encrypted state fluctuation via timer queues and context manipulation

For a more detailed overview of the current state of memory scanning evasion I would recommend Kyle Avery's DEFCON30 talk - Avoiding Memory Scanners: Customizing Malware to Evade YARA, PE-sieve, and More.

Evasion recap



Kyle Avery - Avoiding Memory Scanners: Customizing Malware to Evade YARA, PE-sieve, and More

https://forum.defcon.org/node/241824

And evasion works.

Evasion recap

	Protection	Encryption	Trigger	Bootstrap
Gargoyle	✅		APC timer	ROP
obfuscate-and-sleep	☑	✅	Sleep	shellcode
FOLIAGE	✅	✅	APC timer	CONTEXT
Shellcode Fluctuation	✅		Exception	shellcode
DeepSleep	✅		Sleep	ROP
Ekko	✅	✅	Timer queue	CONTEXT

Each of the approaches is effectively a variant of Gargoyle with a modified trigger or post-trigger bootstrap.
The original CobaltStrike obfuscate-and-sleep didn't fluctuate the region memory protection – but had to use a suspicious RWX region instead.

So how do we detect these evasions?

Niche memory scanners

- Patriot - anomalous thread CONTEXT structures
- Hunt-Sleeping-Beacons - anomalous Wait call stacks
- TickTock - anomalous timer-queue timers

There are some niche memory scanners that can detect each specific variant because of the trigger and/or the bootstrap.

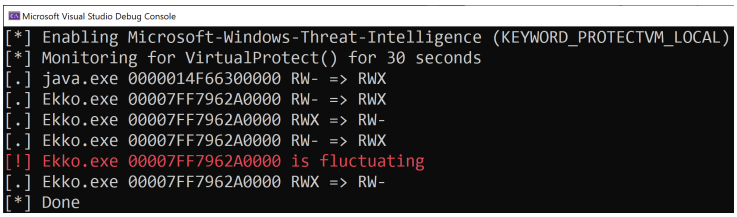
And, as each approach has been detected, offensive researchers have quickly published alternatives and flagged that further variants exist.

So can we detect this class of techniques more directly?

VirtualProtect is the choke point…

Immutable code page principle violations

- Once code pages are written they should never change.
- The memory protection progression for code pages should only be RW to RX.
- Microsoft-Windows-Threat-Intelligence PROTECTVM_LOCAL ETW events
- `IsExecutable(LastProtectionMask) && !IsExecutable(ProtectionMask)`
- (Optionally) Anomalous call stack detection



And for that there are Microsoft-Windows-Threat-Intelligence PROTECTVM ETW events and they include both ProtectionMask and LastProtectionMask. So we can reliably monitor these calls without user-mode hooks.

Technically speaking these events are only available to security products. Pat Hogan's Sealighter-TI project was the first project to make these events accessible to researcher – using now patched PPL exploit. But if you accidentally bring-your-own-vulnerable-driver like my tool then you too can see these events.

We have the events – but what is anomalous?

Firstly there is the caller. ETW events can optionally include call stack traces which can be inspected for anomalies like ROP, indirect calls, or calls originating from unbacked or untrusted code. Such enrichments are currently fragile because Microsoft relies on the user-mode stack to generate these – and this is susceptible to tampering. Microsoft has the opportunity to harden ETW call stacks by instead querying the CET Shadow Stack, though this relies on hardware support.

And secondly, once code pages are written they should never change.
That is, the the memory protection progression for code pages should only be RW to RX.

So we can simply generate an alert whenever executable memory is changed to non-executable – but there are some false positives that we need to deal with.
Some JIT engines unfortunately re-use memory allocations.
And, to be thorough, we should also watch when any executable memory fluctuates from non-writable to writable.
And then there is a smattering of API hooking to account for.

But mostly we should only alert if this is done more than once. This drastically reduces the false positive rate.

This, however, requires maintaining a map of addresses that have been previously executable – since the kernel doesn't store this information for us. Or does it?

An interesting discovery

Gabriel Landau

I think I just made an interesting discovery. If you VirtualAlloc(RWX), it modifies the CFG bitmap accordingly. If you then VirtualAlloc(RW), the CFG bitmap stays as-is.

Gabriel Landau

We may be able to use this to find DLL hollowing and gargoyle-style?

Gabe noticed that Control Flow Guard bitmap entries for an address persist beyond the executable lifetime of that address. And we sat down to see what we could do with this.

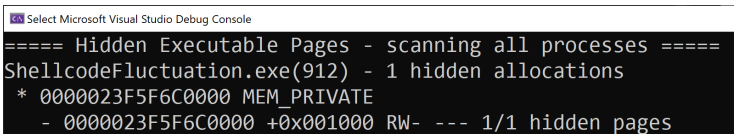
Control Flow Guard bitmap recap

- Time efficient lookup of valid indirect call targets
- One bitmap per process
- Each 2 bits corresponds to 16 virtual addresses
- x64 bitmap is 2TB – mostly shared or reserved
- PE files bring their own bitmap
- Copied to the correct offset in process bitmap during image load
- Permissive backwards compatibility for JIT
- Memory manager simply marks all executable private addresses as valid targets

A quick recap of CFG bitmap and its purpose.
It was designed so that no code changes were required – just recompilation by a CFG-aware toolchain.

CFG bitmap anomalies

- The VAD tree only stores original protection and current protection.
- The CFG bitmap (inadvertently) records the location of all private memory addresses that are, **or have previously been**, executable during the lifetime of the process.
- This can be used to flag memory regions that have been changed from executable to non-executable.



In the second tool, we utilise Gabe's observation that the CFG bitmap is not updated when non-image memory is toggled from executable to non-executable.

We use some undocumented structures to find the CFG bitmap in a process. And then we search for CFG pages that have all bits set, where the corresponding VAD entry is not executable. This makes protection fluctuation stand out beautifully – in most host processes.

Unfortunately, the CFG bitmap is also not updated if the memory is freed.
In particular, JIT compilers (especially .NET) leave similar anomalies in the CFG bitmap, so false positives are possible in some processes where the freed memory is later used for an alternate purpose.

This means that if you are resident in the right process, it is possible to craft a memory layout that appears similar to a false positive.

There is also a stricter opt-in version for newer JIT engines to take advantage of – though I'm only seen Microsoft's chakra do so. JIT memory can be allocated with PAGE_TARGETS_INVALID - and this could bypass the current implementation of my detection.

Now I could make some modifications to find all private CFG pages – and flag the entire corresponding 256KB of memory as suspicious. This would increase the false positive rate significantly though.

Alternatively, I could just monitor ETW-TI for use of that flag. And if you are not chakra…

Evasion opportunities

- Protection fluctuation approaches are actually quite noisy.

- Hide your code pages in plain sight.
- Obfuscate them against current signatures ahead of time.
- Encrypt your data pages when not in use.

- Or launch in a new process every time.
- Scheduled Tasks etc.

So are there still evasion opportunities?

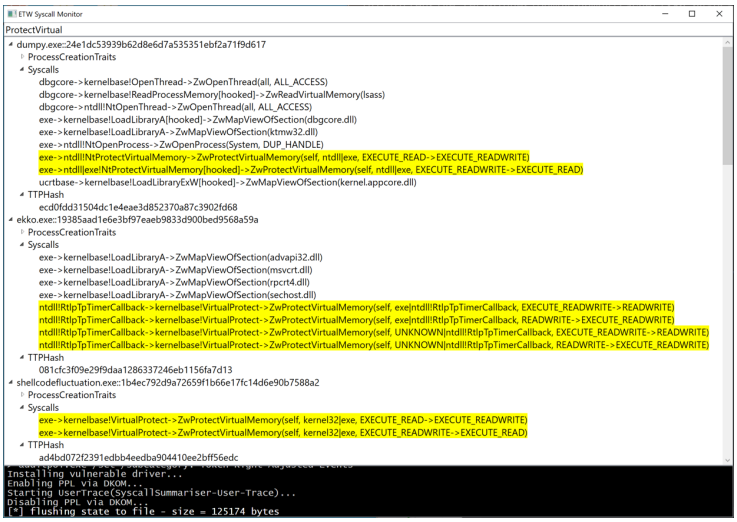
These just-in-time code obfuscation “fluctuation” approaches are actually noisy.
There are a lot of VirtualProtect calls – as well as footprints in the CFG bitmap.

Jumping to a new allocation on each execution might help - but the old allocations will leave cumulative footprints in the CFG.

Hiding in a process that is already routinely re-writing executable memory is possible - these are rare though.

Instead, less is more.
You can still encrypt your data pages when not in use – and many YARA signatures are actually on data not code.
And for your code, just obfuscate it against current signatures ahead of time.
This is same approach as on-disk AV evasion – though you can’t rely on packers.

Hunting via process behaviour summaries



What I'm about to describe is not a detection. It is an approach to telemetry collection that, without prior technique knowledge, would still have flagged the evasion as new behaviour requiring investigation.

Even if you hide your user-mode shellcode perfectly, it will likely need to interact with the kernel eventually. The only way to truly hide is if every action you take is normal for the host process. But how do we find abnormal at scale?

Most (academic) approaches to dynamic malware detection utilise full syscall traces from instrumented sandbox environments. This is far too verbose for general production use.

Most current EDR solutions effectively produce syscall trace subsets. They only send high value telemetry - especially process and network events. Some techniques cannot be detected from these subsets.

So rather than outputting 1000s or 100s of detailed events – can we just output just 10s of behaviours?

We're probably all familiar with a binaries file's Import Table and the usefulness of ImpHash for pivoting to similar samples.
You can often roughly infer the purpose of software just from its Import Table.
Some of you are probably also familiar with Mandiant CAPA – which does automated static analysis to pull out calls with parameters and map these onto capabilities. This takes the approach a step further.

The problem with Import Table analysis is GetProcAddress. That is, software can resolve functions dynamically.

But the kernel doesn't care about GetProcAddress. Irrespective of whether it was resolved statically or dynamically the only way to make a syscall is to make that syscall.

A complementary approach to volume minimisation is to map all the visible syscall invocations onto a small finite set and to only log when observed for the first time. This can be done by either discarding or normalising any parameter fields with high cardinality. For example in a VirtualProtect event the lpAddress parameter could be normalised to page type (image|non-image) and dwSize can be discarded.

Doing this at runtime, provides a mechanism to scale collection for otherwise high volume telemetry sources.

This is the third tool. It provides a way to explore the high levels behaviours of a process through very concise syscall summaries.

This collection wouldn't detect unknown techniques as immediately malicious - but it would detect that something abnormal had occurred. And threat hunters (supported by automated triage) can follow this crumb.
And, if you are storing these summaries, it would allow retrospective detection once the technique is uncovered and understood.

I personally believe that this could be the basis of a VirusTotal for behaviour.

Black Hat Sound Bytes

- Threat-Intelligence ETW can be used to detect violations of the immutable code page principle.
- The CFG bitmap can be used to detect shellcode hidden at a point-in-time via changed memory protections such as Gargoyle.
- Kernel telemetry can be used to construct process behaviour summaries – which can be used to identify behavioural outliers for more detailed investigation.

Memory scanning is a useful layer of defence.

In the usual cat-and-mouse of cybersecurity, our offensive researchers have helped defenders find its detection gaps and we've responded with improvements.

Today I've demonstrated that Gargoyle-esque techniques that have long been relied on can be detected in two ways and discovered via a third.

Black Hat Sound Bytes

- Threat-Intelligence ETW can be used to detect violations of the immutable code page principle.
- The CFG bitmap can be used to detect shellcode hidden at a point-in-time via changed memory protections such as Gargoyle.
- Kernel telemetry can be used to construct process behaviour summaries – which can be used to identify behavioural outliers for more detailed investigation.

- But, without intervention from Microsoft, private executable memory will likely remain an indefensible boundary for kernel-mode security products.

But detection != prevention.

Microsoft's current architecture does not yet provide security products with sufficient opportunities to in-memory threats like CobaltStrike from running. And I personally think that it has passed the in-the-wild prevalence test for servicing.

In the meantime, security vendors will continue to layer on defence-in-depth approaches like memory scanning and scalable anomaly detection.

Questions

Tools

- https://github.com/jdu2600/EtwTi-FluctuationMonitor
- https://github.com/jdu2600/CFG-FindHiddenShellcode
- https://github.com/jdu2600/Etw-SyscallMonitor

Detection References

- https://github.com/VirusTotal/yara
- https://github.com/hasherezade/pe-sieve
- https://github.com/forrest-orr/moneta
- https://www.elastic.co/security-labs/hunting-memory
- https://www.elastic.co/blog/detecting-cobalt-strike-with-memory-signatures
- https://github.com/joe-desimone/patriot
- https://github.com/thefLink/Hunt-Sleeping-Beacons
- https://github.com/WithSecureLabs/TickTock

Evasion References

- https://github.com/JLospinoso/gargoyle
- https://www.cobaltstrike.com/blog/cobalt-strike-3-12-blink-and-youll-miss-it/
- https://github.com/realoriginal/foliage
- https://github.com/mgeeky/ShellcodeFluctuation
- https://github.com/thefLink/DeepSleep
- https://github.com/Cracked5pider/Ekko
- https://www.blackhillsinfosec.com/avoiding-memory-scanners/

OS References

- https://en.wikipedia.org/wiki/W%5EX
- https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect
- https://github.com/jdu2600/Windows10EtwEvents/blame/master/manifest/Microsoft-Windows-Threat-Intelligence.tsv
- https://www.elastic.co/security-labs/finding-truth-in-the-shadows