Malware Morphology

Golden Tickets

Golden Tickets

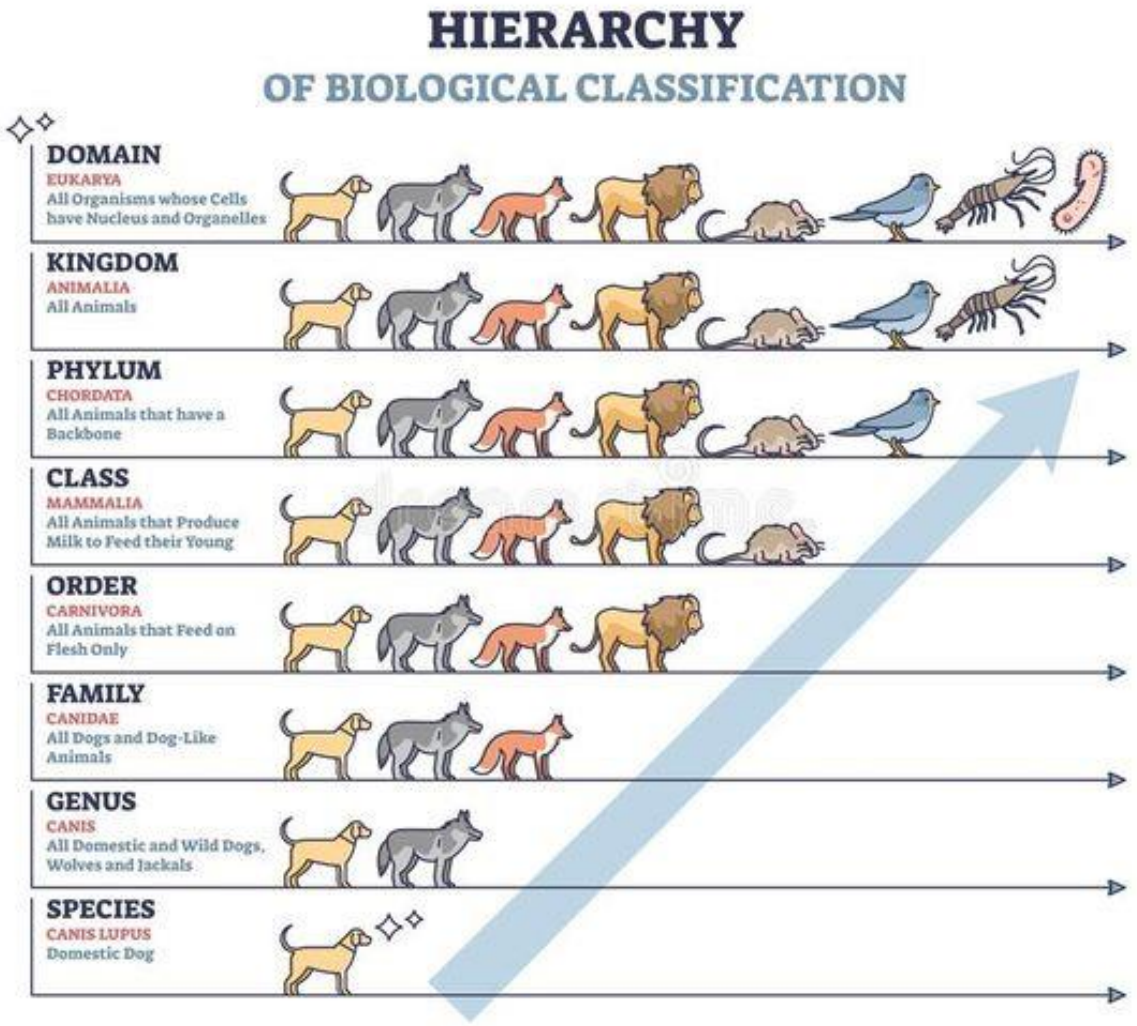
Intro to Morphology

Morphology

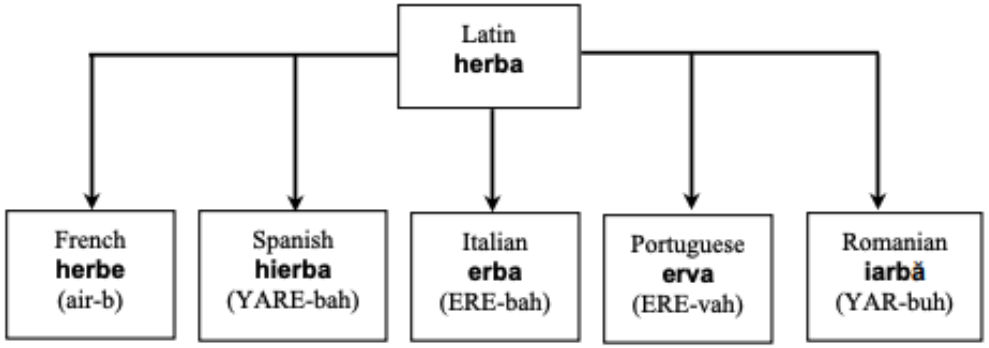
- The study of the form and structure of things.
  - Biology – The form and structure of living organisms (animals, plants, etc.)
  - Linguistics – The study of words, how they are formed.
- Anatomy is one sub-discipline of biological morphology.
- A primary tool used to measure the similarity of two things.
- This workshop intends to demonstrate how morphological analysis of malware samples is useful in many Detection and Response disciplines.
  - Teach a process for morphological analysis of malware samples.
  - Demonstrate how to categorize samples into a hierarchical taxonomy.
  - Explain implications of similarity on Detection and Response.

Biology

- Organisms are generally organized scientifically in something called the Linnaeus Taxonomy.
  - This organization is primarily based on their morphological features.
- Each level in the taxonomy is called a "rank."
  - Species, Genus, Family, Order, etc.
- The general idea is that the lower the rank in which two particulars converge, the more similar they are.
  - Canis Familaris vs. Canis Lupus
  - Canis Familaris vs. Felis Catus



Linguistics

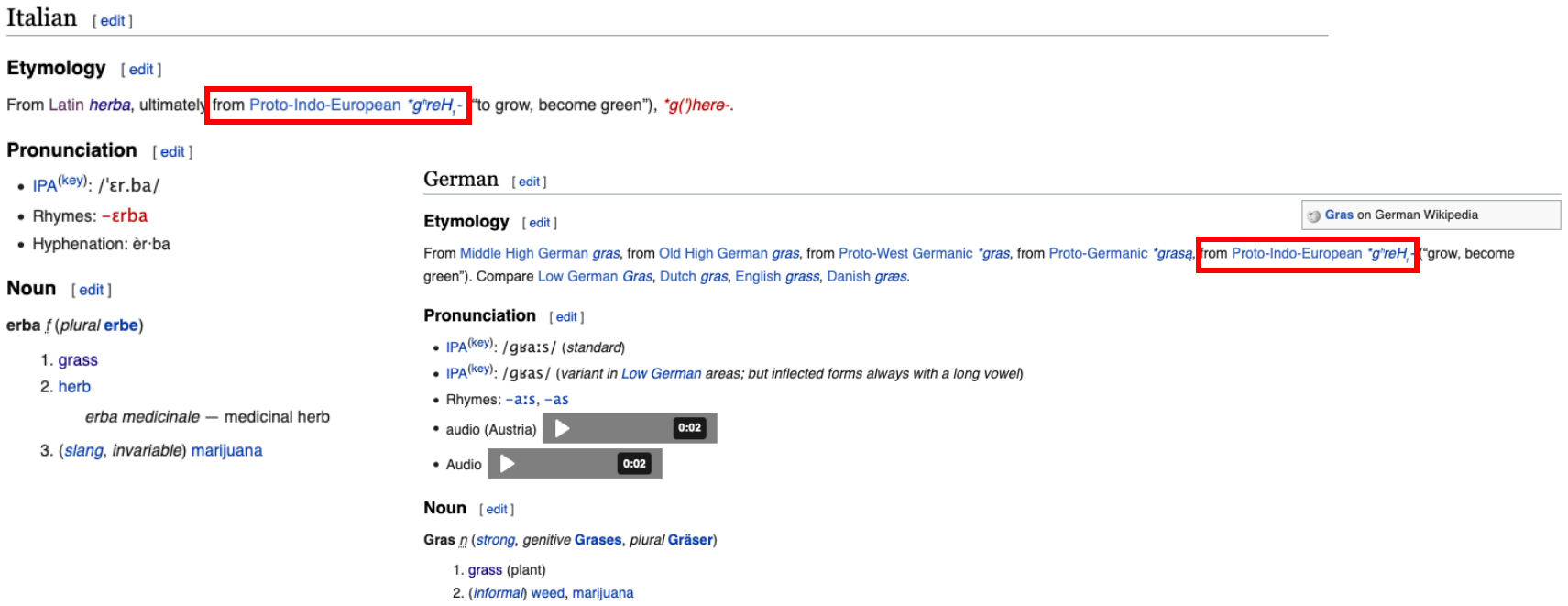


1. All the languages dropped the *h*—the spellings in French and Spanish maintain it, just as English spelling maintains the "silent" *e*.
2. *Moderate changes*. Italian is one of the closest Romance languages to Latin, and other than the lost *h*, it preserves the word intact. French goes somewhat further and drops the final *-a* as well. Spanish keeps this but changes the *e* to an *ie* (pronounced "yeh"), while Portuguese instead softens the *b* to a *v*.
3. *Radical changes*. Romanian doesn't just insert a *y* sound before the *e* as Spanish does but has a whole new sound *ia* (pronounced "yah"), and the symbol over the final *-a* indicates that this is a new sound, roughly "uh." Consider that similar changes happen to every word in the language, and it is easy to see how one language becomes several new ones.

- In Linguistics, morphology helps to classify similarities between words and languages (phonetic).
- Most to Least Similar to herba:
  - IT – "erba"
  - PT – "erva"
  - FR – "herbe"
  - SP – "hierba"
  - RO – "iarbă"
- Less Similar
  - DE – "Gras", NO – "gress", EN – "grass"
  - KO – "잔디" (jandi)

Common Ancestor

- Even though Italian's "erba" and German's "Gras" appear VERY different, both words can be traced back to Proto-Indo-European's "g^hreH"



Italian [edit]

Etymology [edit]

From Latin *herba*, ultimately from Proto-Indo-European **gʰreH₁-* ("to grow, become green"), **g(')herə-*.

Pronunciation [edit]

- IPA(key): /ˈɛr.ba/
- Rhymes: –ɛrba
- Hyphenation: èr·ba

Noun [edit]

erba *f* (*plural* **erbe**)

1. grass
2. herb

   erba medicinale — medicinal herb
3. (*slang*, *invariable*) marijuana

German [edit]

Gras on German Wikipedia

Etymology [edit]

From Middle High German *gras*, from Old High German *gras*, from Proto-West Germanic **gras*, from Proto-Germanic **grasą*, from Proto-Indo-European **gʰreH₁-* ("grow, become green"). Compare Low German *Gras*, Dutch *gras*, English *grass*, Danish *græs*.

Pronunciation [edit]

- IPA(key): /ɡʁaːs/ (*standard*)
- IPA(key): /ɡʁas/ (*variant in Low German areas; but inflected forms always with a long vowel*)
- Rhymes: –aːs, –as
- audio (Austria) 0:02
- Audio 0:02

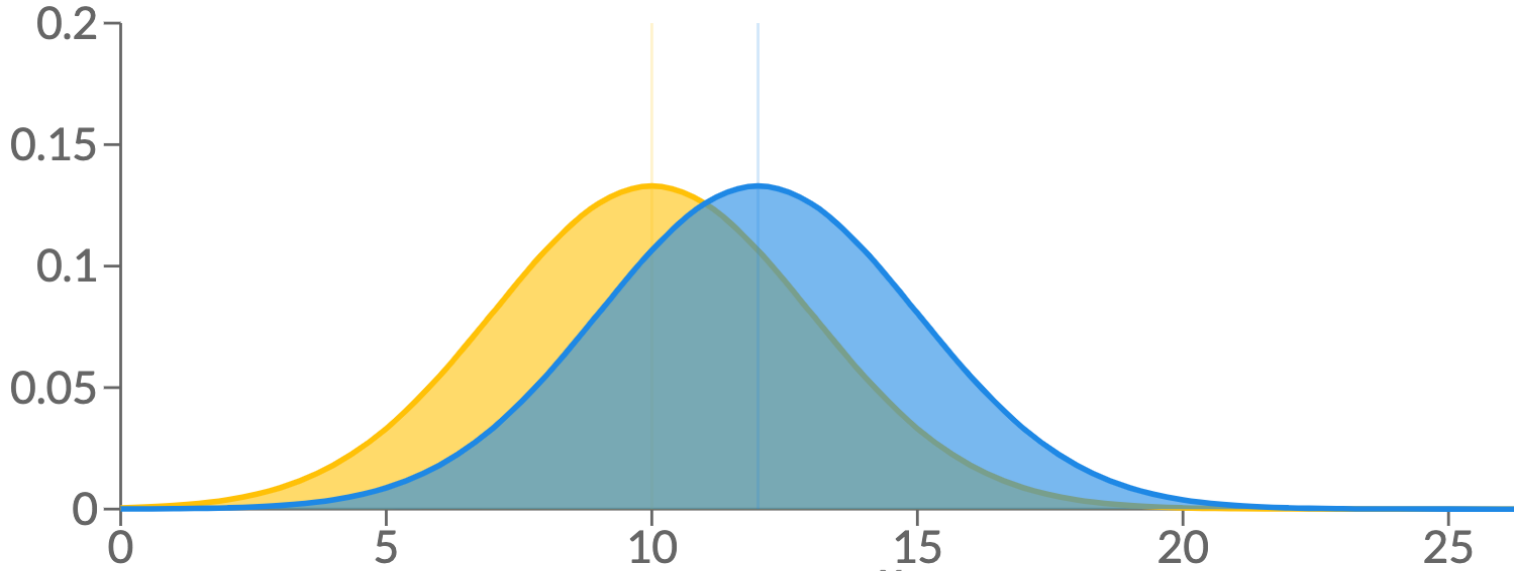
Noun [edit]

Gras *n* (*strong*, *genitive* **Grases**, *plural* **Gräser**)

1. grass (plant)
2. (*informal*) weed, marijuana

Folk Taxonomies vs. Scientific Taxonomies

- Folk vs. Scientific Taxonomy
  - The overlap is tremendous
  - Discontinuities arise at the edge of categories.
    - Are **BATS** better grouped with **BIRDS** or **MAMMALS**?
    - Are **WHALES** better grouped with **FISH** or **MAMMALS**?



Fundamental Questions

How similar are sc /create, New-Service, and SharpSC

What fills the **gap** between (Sub-)Techniques and Tools?

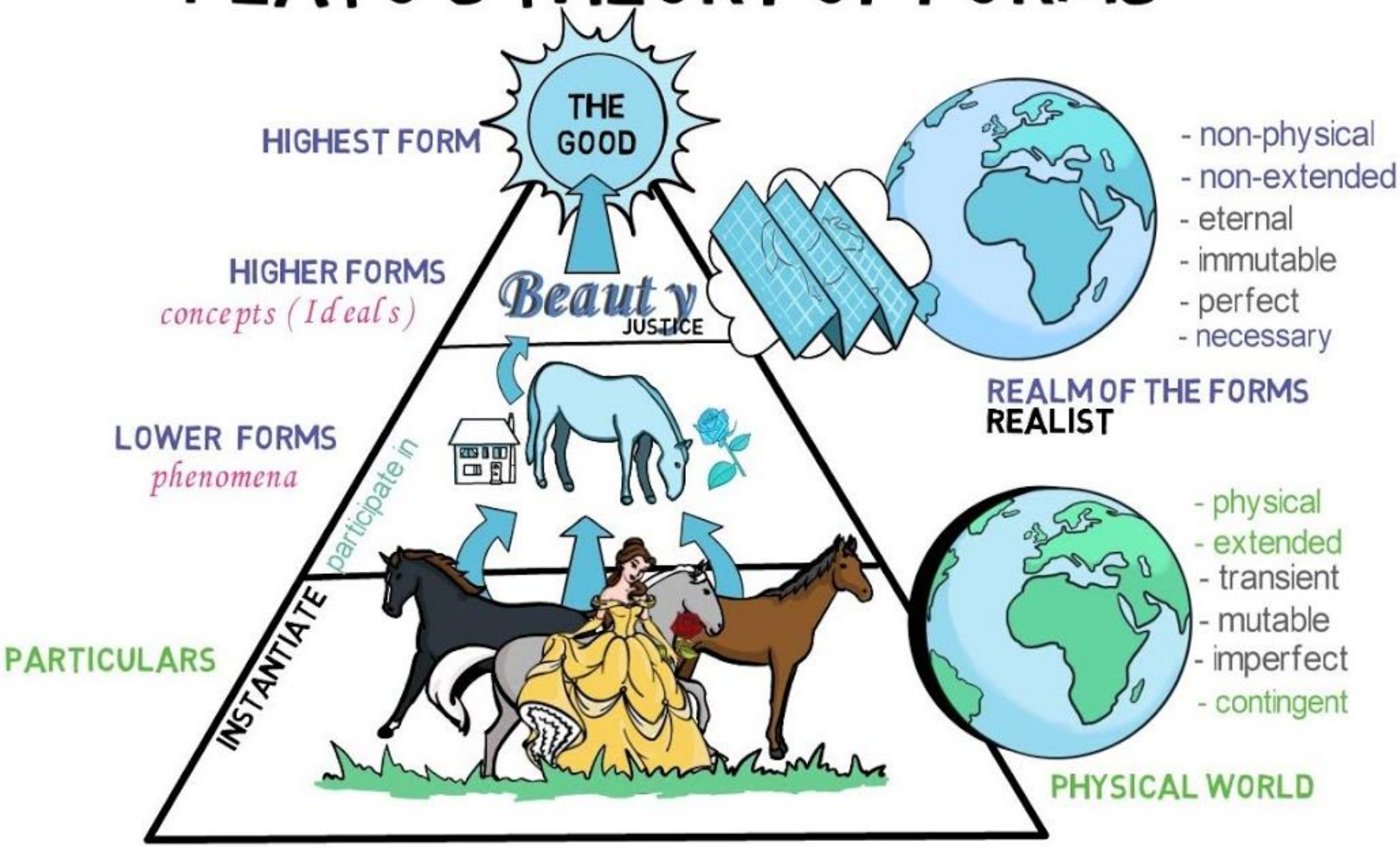
Are there degrees of similarity?

Similarity – Literally or Conceptually

- What does it mean for two malware samples to be the SAME?
- How can we measure similarity?
  - Cryptographic Hashes (MD5, SHA1, SHA256)
    - Only measure ABSOLUTE similarity
    - There's no way to determine if one bit changed or if the entire sample is different.
  - Piecewise and Fuzzy Hashing[1]
    - Generate traditional hash, but also generate hash values for segments of files.
    - This assumes that changes will be localized to certain locations (change mimikatz to mimidogz).
  - Imphash[2]
    - Idea that Portable Executables that import the same API functions are probably similar in function despite changes to less significant bits.
    - Still lacks ability to distinguish between small and large changes.

1. https://www.sciencedirect.com/science/article/pii/S1742287606000764?via%3Dihub
2.https://www.mandiant.com/resources/blog/tracking-malware-import-hashing

PLATO'S THEORY OF FORMS



https://youtu.be/TissHzEG_JY

The Map is Not the Territory

- Concept posited by Alfred Korzybski.
  - "A map is not a territory it represents, but, if correct, it has a similar structure to the territory, which accounts for its usefulness."
- Our model is necessarily a low-resolution abstraction of reality.
- The map is an explicit representation of our understanding.
  - There is a significant difference between implicit and explicit maps.
- So how do we begin to build our map and what level of resolution is important, or even necessary, for our purpose?
  - We can begin to approximate the form (the technique) through analyzing the particulars (tools)

Application of Morphology

- Detection Engineers
  - Helps to determine detection analytic scope based on behavioral differences between variations of a technique.
- Red Teamers
  - Facilitates tradecraft decision making where alternative tools can be selected for the purpose of evading specific evidence or telemetry capabilities.
- Cyber Threat Intel Analysts
  - Enables the measurement of similarity between samples and more granular categorization.
- Security Architects
  - Allows for better architectural decisions regarding the configuration of preventative controls.

Tool Analysis

Samples

- Course GitHub Repository
  - https://github.com/jaredcatkinson/MalwareMorphology
- In this class we will analyze six token theft "malware" samples.
  - Sample 1 (C++) - uses SetThreadToken
  - Sample 2 (PowerShell) - uses SetThreadToken
  - Sample 3 (C++) - uses ImpersonateLoggedOnUser
  - Sample 4 (C++) - uses LogonUser + ImpersonateLoggedOnUser
  - Sample 5 (C++) - uses CreateProcessWithToken
  - Sample 6 (C++) - uses CreateProcess + SetThreadToke

Tool Selection Criteria

- The best way to build the map is to explore the territory.
- Most techniques will have many samples to choose from that each include their own implementation spin.
  - The good news is that the first sample will always be novel.
- Use the following criteria to select a sample when first starting out:
  1. Open Source vs. Closed Source
     - Open-source samples removes the burden of reverse engineering or interpreting assembly.
  2. Simple vs. Complex
     - Simple tools implement only one tool and generally take minimal command line arguments (ex. Out-Minidump)
  3. Programming Language (C vs. PowerShell)
     - C is a straightforward language where what you see is what you get.
     - Managed languages, like PowerShell, may act transparently and require much more knowledge in order to interpret what the code does.

What does "Malware Sample" Mean?

- When someone asks the question "what does mimikatz do?" It is not possible to answer that question succinctly.

Three levels of things that can all be considered "malware":

- Stand-alone Tools
  - Does one simple task (dump credentials from LSASS)
  - Ex. New-Service, Out-Minidump
- Complex Tools
  - Has numerous modules that facilitate different features via finite interactions.
    - Ex. sc.exe, mimikatz, rubeus
- C2 Platforms
  - Manages access over the long term and integrates features of other tools.
    - Ex. Cobalt Strike, Mythic

Functions

- An interface that allows developers to execute common routines without having to reinvent the code
- Windows Functions:
  - Win32 APIs – core set of APIs built into the OS that Microsoft prefers developers to interact with
  - Internal Functions – functions that are used internally within code but not exported, so they can't be used externally
  - Native APIs – lower-level APIs that are often called by Win32 APIs to carry out tasks
  - System calls (syscalls) - bridge between user-mode and kernel-mode functionality.
- For this presentation we will not dive into kernel-level functions

Function Chain

- The first layer of abstraction above the tool is the function chain.
- A function chain is the sequence of API functions that are called by the sample.
- Imagine that each byte, in each sample, is interpreted symbolically as an assembly opcode or data.
  - Some opcodes are more significant than others.
  - Branching instructions, like jmp or call, are more significant than nop instructions.
  - By focusing on function calls, we are ignoring insignificant small detail and attending to the most significant bytes.

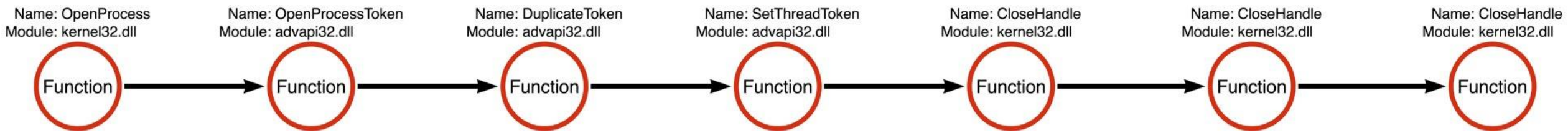
Sample 1: Source Code

```
8         DWORD PID = atoi(argv[1]);
9
10        HANDLE hToken, hDuplicate, hProcess = NULL;
11
12        hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, true, PID);
13        OpenProcessToken(hProcess, TOKEN_DUPLICATE, &hToken);
14        DuplicateToken(hToken, SecurityImpersonation, &hDuplicate);
15        SetThreadToken(NULL, hDuplicate);
16
17        CloseHandle(hDuplicate);
18        CloseHandle(hToken);
19        CloseHandle(hProcess);
```


https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%201/src/Source.cpp

Sample 1: Function Chain

```
8         DWORD PID = atoi(argv[1]);
9
10        HANDLE hToken, hDuplicate, hProcess = NULL;
11
12        hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, true, PID);
13        OpenProcessToken(hProcess, TOKEN_DUPLICATE, &hToken);
14        DuplicateToken(hToken, SecurityImpersonation, &hDuplicate);
15        SetThreadToken(NULL, hDuplicate);
16
17        CloseHandle(hDuplicate);
18        CloseHandle(hToken);
19        CloseHandle(hProcess);
```



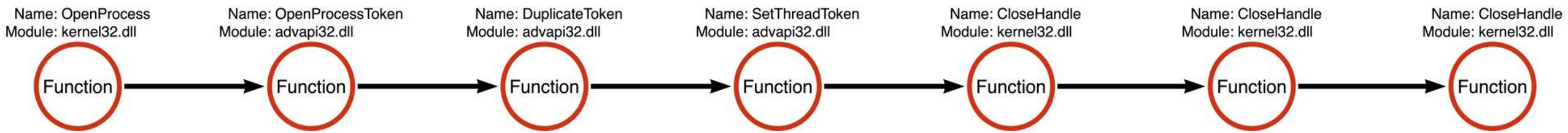
https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%201/src/Source.cpp

Sample 2: Source Code

```
34      $hProcess = OpenProcess -ProcessId $ProcessId -DesiredAccess PROCESS_QUERY_LIMITED_INFORMATION
35      $hToken = OpenProcessToken -ProcessHandle $hProcess -DesiredAccess TOKEN_DUPLICATE
36      $hDupToken = DuplicateToken -TokenHandle $hToken
37      SetThreadToken -Token $hDupToken
38
39      CloseHandle -Handle $hDupToken
40      CloseHandle -Handle $hToken
41      CloseHandle -Handle $hProcess
```

Sample 2: Function Chain

```
34          $hProcess = OpenProcess -ProcessId $ProcessId -DesiredAccess PROCESS_QUERY_LIMITED_INFORMATION
35          $hToken = OpenProcessToken -ProcessHandle $hProcess -DesiredAccess TOKEN_DUPLICATE
36          $hDupToken = DuplicateToken -TokenHandle $hToken
37          SetThreadToken -Token $hDupToken
38
39          CloseHandle -Handle $hDupToken
40          CloseHandle -Handle $hToken
41          CloseHandle -Handle $hProcess
```

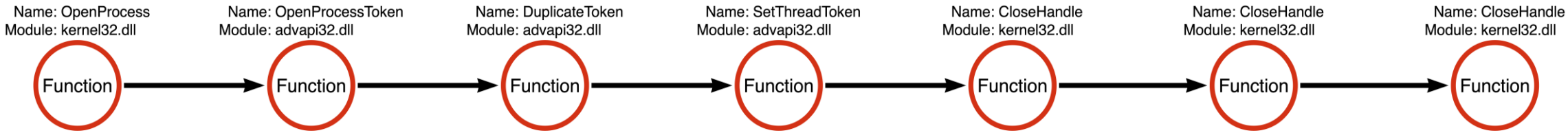


https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%202/src/Set-ThreadToken.ps1

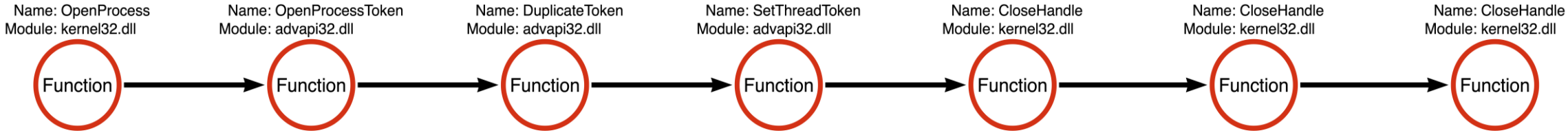
Literally Different, but Functionally Equivalent

- Different tools, different programming languages, same functions.
- These differences between these samples are essentially limited to small details such as variable names.
- A behavioral detection is perfect for this scenario.

Function Chain – Sample 1 and 2



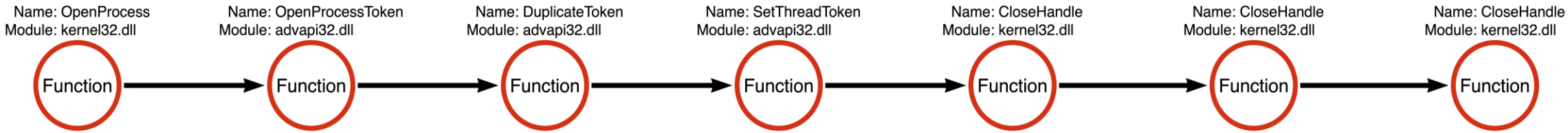
Function Chain – Sample 3



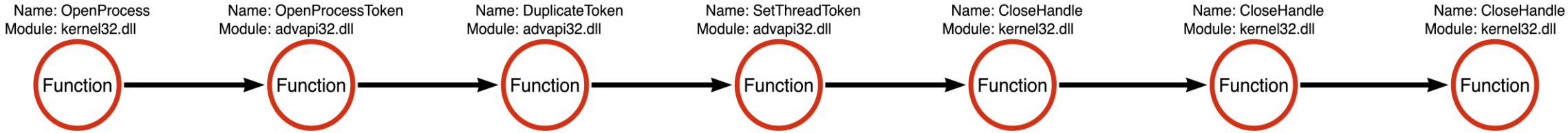
Literally Different, but Functionally Equivalent

- Different tools, different programming languages, same functions.
- These differences between these samples are essentially limited to small details such as variable names.
- A behavioral detection is perfect for this scenario.

Function Chain – Sample 1 and 2



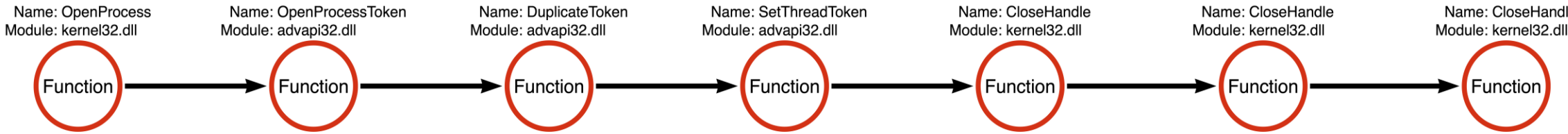
Function Chain – Sample 3



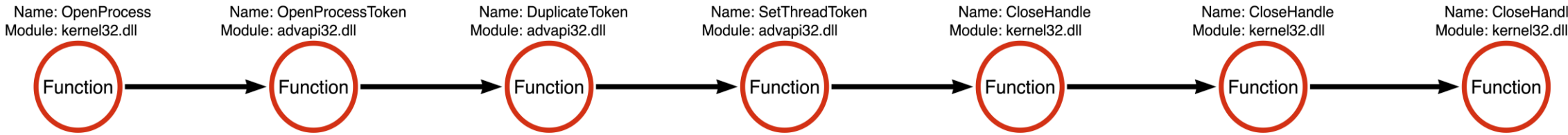
Literally Different, but Functionally Equivalent

- Different tools, different programming languages, same functions.
- These differences between these samples are essentially limited to small details such as variable names.
- A behavioral detection is perfect for this scenario.

Function Chain – Sample 1 and 2



Function Chain – Sample 3



Lab 1: Function Chain

Analyzing Sample 3 to understand how it is performing impersonation. Analyzing the tool helps identify the capabilities the code is leveraging to be successful.

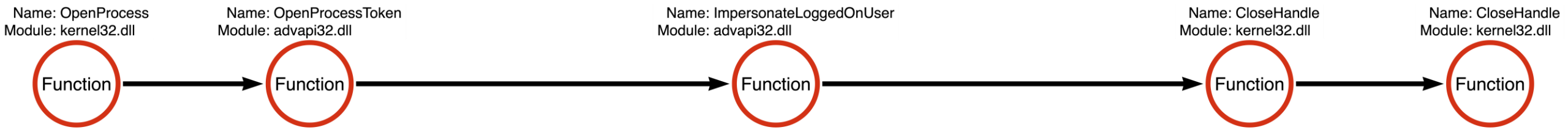
Sample 3: Source Code

```
8         DWORD PID = atoi(argv[1]);
9
10        HANDLE hToken, hProcess = NULL;
11
12        hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, true, PID);
13        OpenProcessToken(hProcess, TOKEN_DUPLICATE | TOKEN_QUERY, &hToken);
14        ImpersonateLoggedOnUser(hToken);
15
16        CloseHandle(hToken);
17        CloseHandle(hProcess);
18
19        return 0;
```


https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%203/src/Source.cpp

Sample 3: Function Chain

```
8         DWORD PID = atoi(argv[1]);
9
10        HANDLE hToken, hProcess = NULL;
11
12        hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, true, PID);
13        OpenProcessToken(hProcess, TOKEN_DUPLICATE | TOKEN_QUERY, &hToken);
14        ImpersonateLoggedOnUser(hToken);
15
16        CloseHandle(hToken);
17        CloseHandle(hProcess);
18
19        return 0;
```

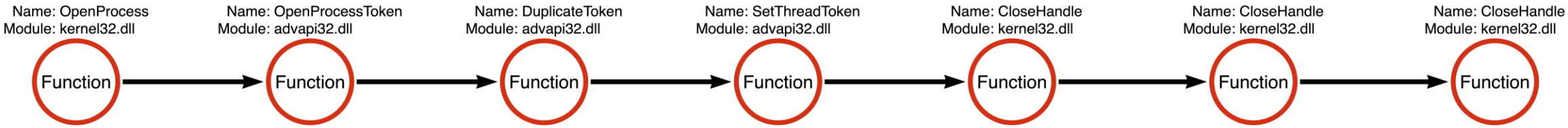


https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%203/src/Source.cpp

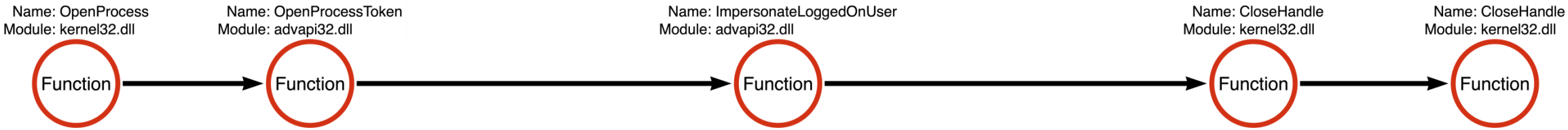
Literally Different, and Functionally Different

- The DuplicateToken -> SetThreadToken -> CloseHandle function sequence is replaced by ImpersonateLoggedOnUser.
- This indicates that API hooking approaches may fail.
- What makes ImpersonateLoggedOnUser different than SetThreadToken?

Function Chain – Sample 1 and 2



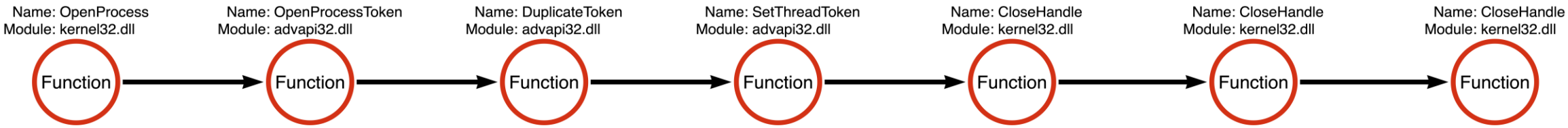
Function Chain – Sample 3



Literally Different, and Functionally Different

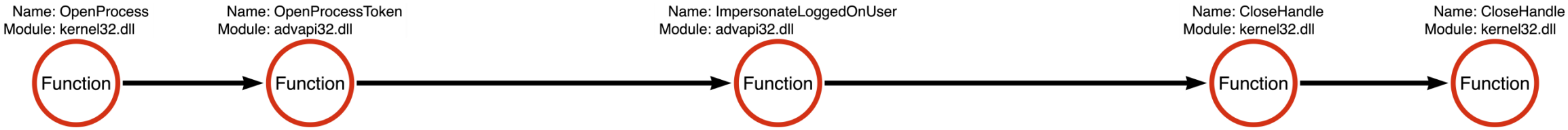
- The DuplicateToken -> SetThreadToken -> CloseHandle function sequence is replaced by ImpersonateLoggedOnUser.
- This indicates that API hooking approaches may fail.
- What makes ImpersonateLoggedOnUser different than SetThreadToken?

Function Chain – Sample 1 and 2



Function Chain – Sample 3

?



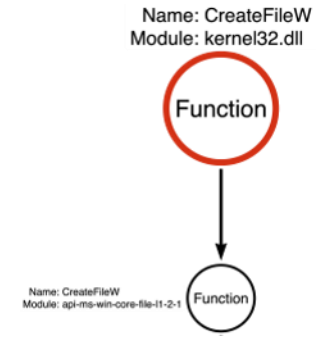
Function Call Stack

Name: CreateFileW
Module: kernel32.dll

Win32 API Function

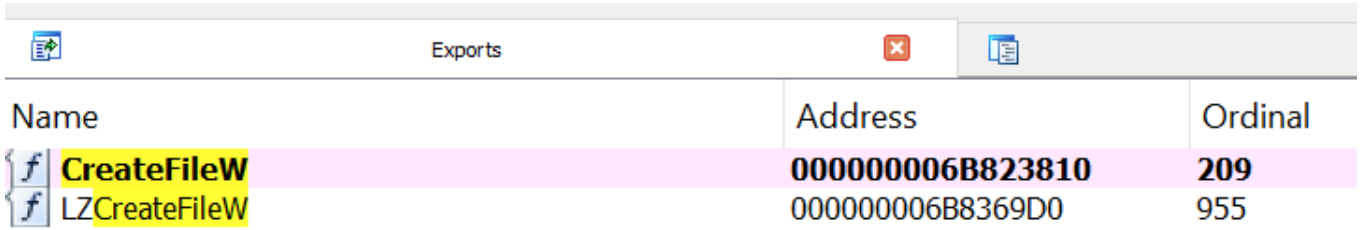
- Highest level API that Microsoft exposes
- Microsoft prefers that these functions are called as they will always have support, the underlying/undocumented function may not

```
; Exported entry 209. CreateFileW


; Attributes: thunk

; HANDLE __stdcall CreateFileW(LPCWSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile)
public _CreateFileW@28
_CreateFileW@28 proc near

lpFileName= dword ptr  4
dwDesiredAccess= dword ptr  8
dwShareMode= dword ptr  0Ch
lpSecurityAttributes= dword ptr  10h
dwCreationDisposition= dword ptr  14h
dwFlagsAndAttributes= dword ptr  18h
hTemplateFile= dword ptr  1Ch

jmp     ds:__imp__CreateFileW@28 ; CreateFileW(x,x,x,x,x,x,x)
_CreateFileW@28 endp
```

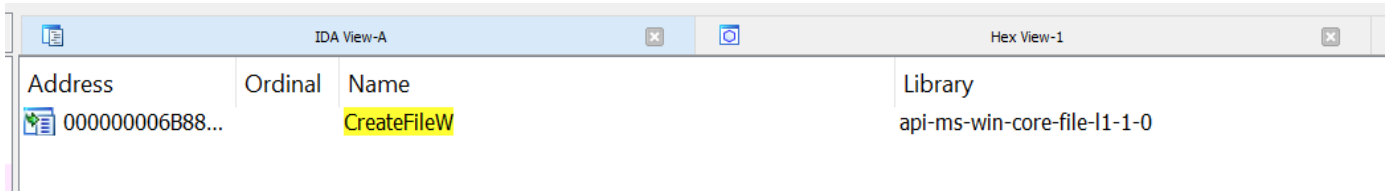


Imported and Exported Functions

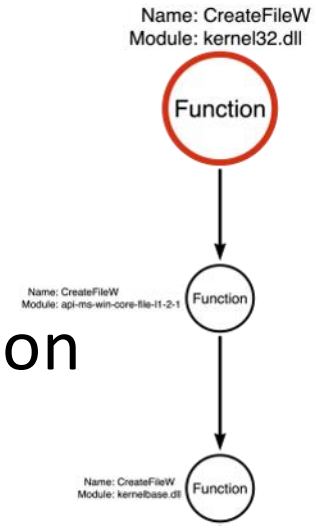
- Exported Functions
  - Functions that want to expose their capability outside their current PE.
  - Function definitions can be found for an exported function within that same PE.



- Imported Functions
  - Functions that are used within the code, but their definition is not within that same PE



API Sets



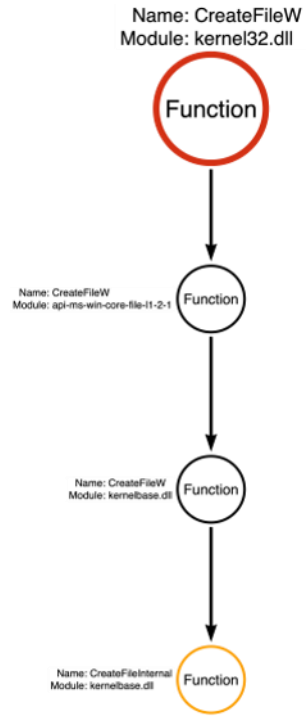
- Query mechanism that redirects APIs to their actual implementation
  - Across systems different API implementations are available in different DLLs/EXEs, API Set points to the correct one for that system to reduce complexity
- Detect if an API is exposed on the current system or not

```
PS C:\Users\Administrator> Get-NtApiSet -Name api-ms-win-core-file-l1-2-1 | select -ExpandProperty hosts

ImportModule HostModule     DefaultHost
------------ ----------     -----------
             kernelbase.dll        True
```


https://patentimages.storage.googleapis.com/d0/5f/79/2583d676e6c0de/US20140366045A1.pdf

Internal Function

- Neither an exported or imported function
- Can not be accessed outside of its designated PE (DLL/EXE)

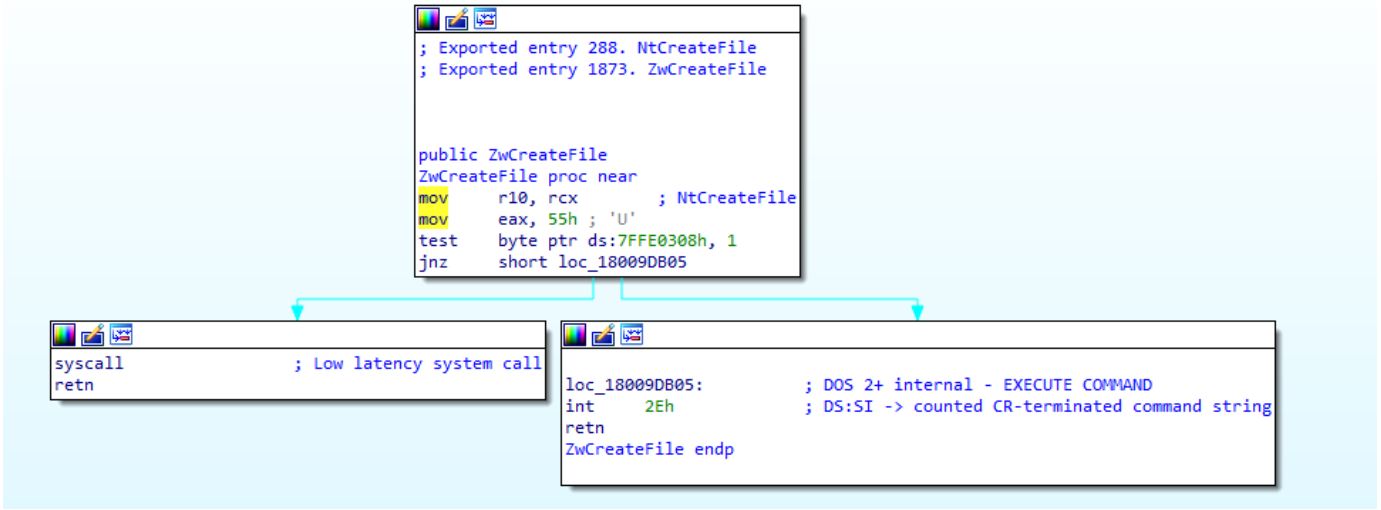


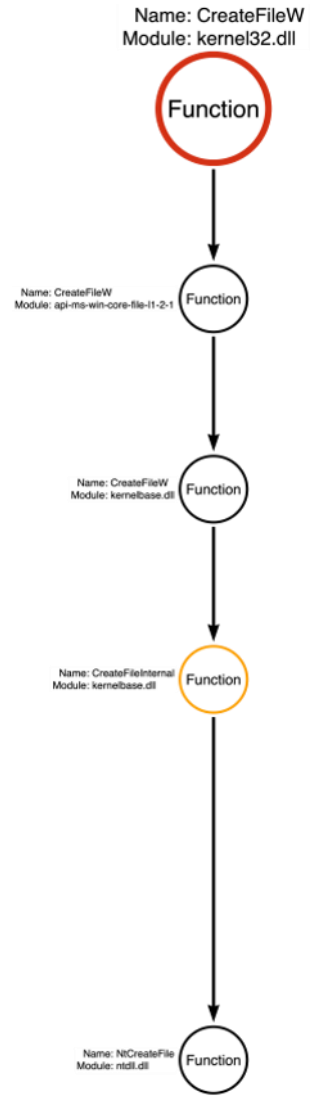

```
; Attributes: bp-based frame fpd=60h

CreateFileInternal proc near

AllocationSize= qword ptr -140h
FileAttributes= dword ptr -138h
ShareAccess= dword ptr -130h
CreateDisposition= dword ptr -128h
CreateOptions= dword ptr -120h
EaBuffer= qword ptr -118h
EaLength= dword ptr -110h
```

```
loc_180036529:
mov     eax, [rsp+160h+var_100]
lea     r9, [rbp+60h+IoStatusBlock] ; IoStatusBlock
mov     [rsp+160h+EaLength], edi ; EaLength
lea     r8, [rbp+60h+ObjectAttributes] ; ObjectAttributes
mov     [rsp+160h+EaBuffer], rbx ; EaBuffer
lea     rcx, [rbp+60h+FileHandle] ; FileHandle
mov     [rsp+160h+CreateOptions], eax ; CreateOptions
and     r14d, 5AFFA7h
mov     eax, [rsp+160h+var_F0]
or      r13d, 100080h
mov     [rsp+160h+CreateDisposition], r12d ; CreateDisposition
mov     edx, r13d       ; DesiredAccess
mov     [rsp+160h+ShareAccess], eax ; ShareAccess
mov     [rsp+160h+FileAttributes], r14d ; FileAttributes
mov     [rsp+160h+AllocationSize], 0 ; AllocationSize
call    cs:__imp_NtCreateFile
nop     dword ptr [rax+rax+00h]
mov     edi, eax
cmp     eax, 0C0000022h
jnz     short loc_1800365F0
```

Native API Functions

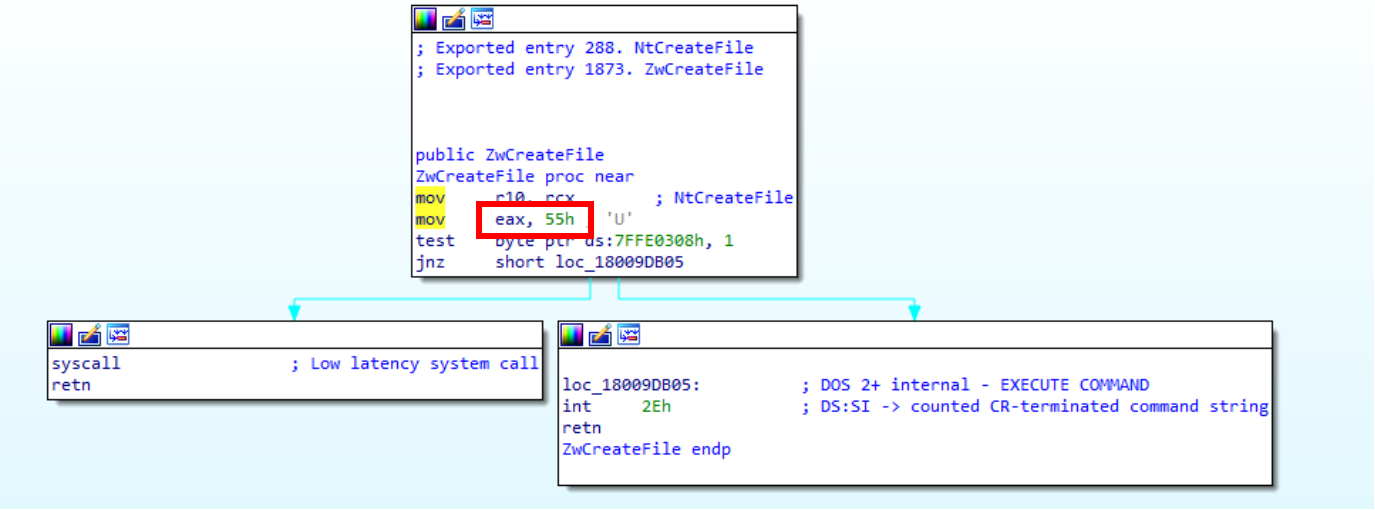
- Last user-mode function before the code transitions into the kernel
- Nt/Zw prefixed functions
- Stored within ntdll.dll/win32u.dll

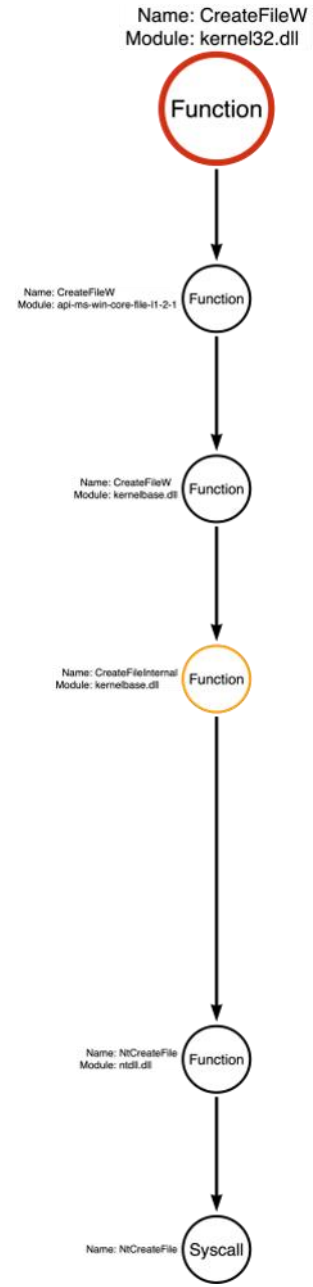




System Calls (syscalls)

- Transitions functionality from user-mode to kernel-mode
- Takes value passed into EAX and passes it to the System Service Dispatch Table (SSDT)



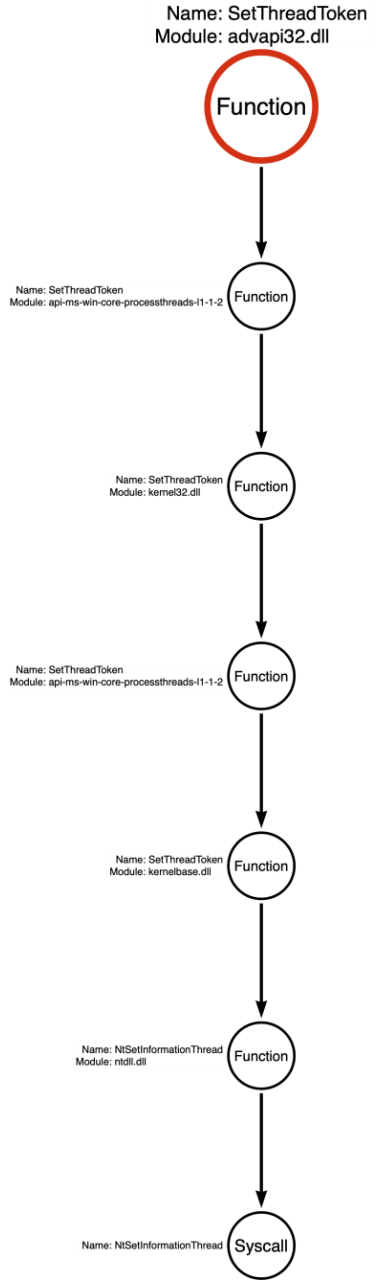


Lab 2: Function Call Stack

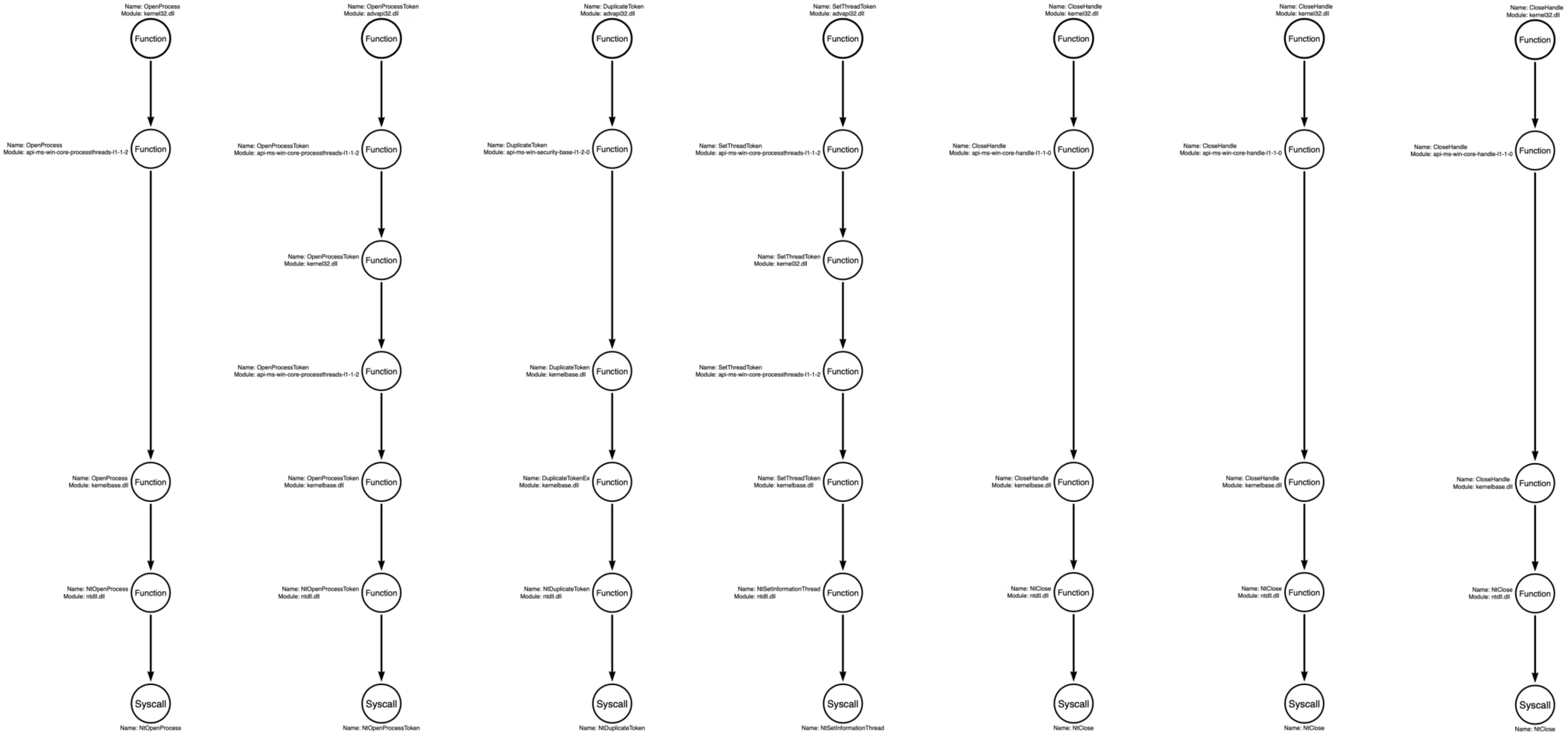
Analyzing the function call stack for the SetThreadToken API function. This function serves as a relevant example of a "simple" function or a function that does one and only one thing. While this lab focuses on a specific function call, the instructions can be used for all additional functions. We recommend practicing this process with some of the other functions in the SetThreadToken function chain (OpenProcess, OpenProcessToken, etc.).

Call Stack - SetThreadToken

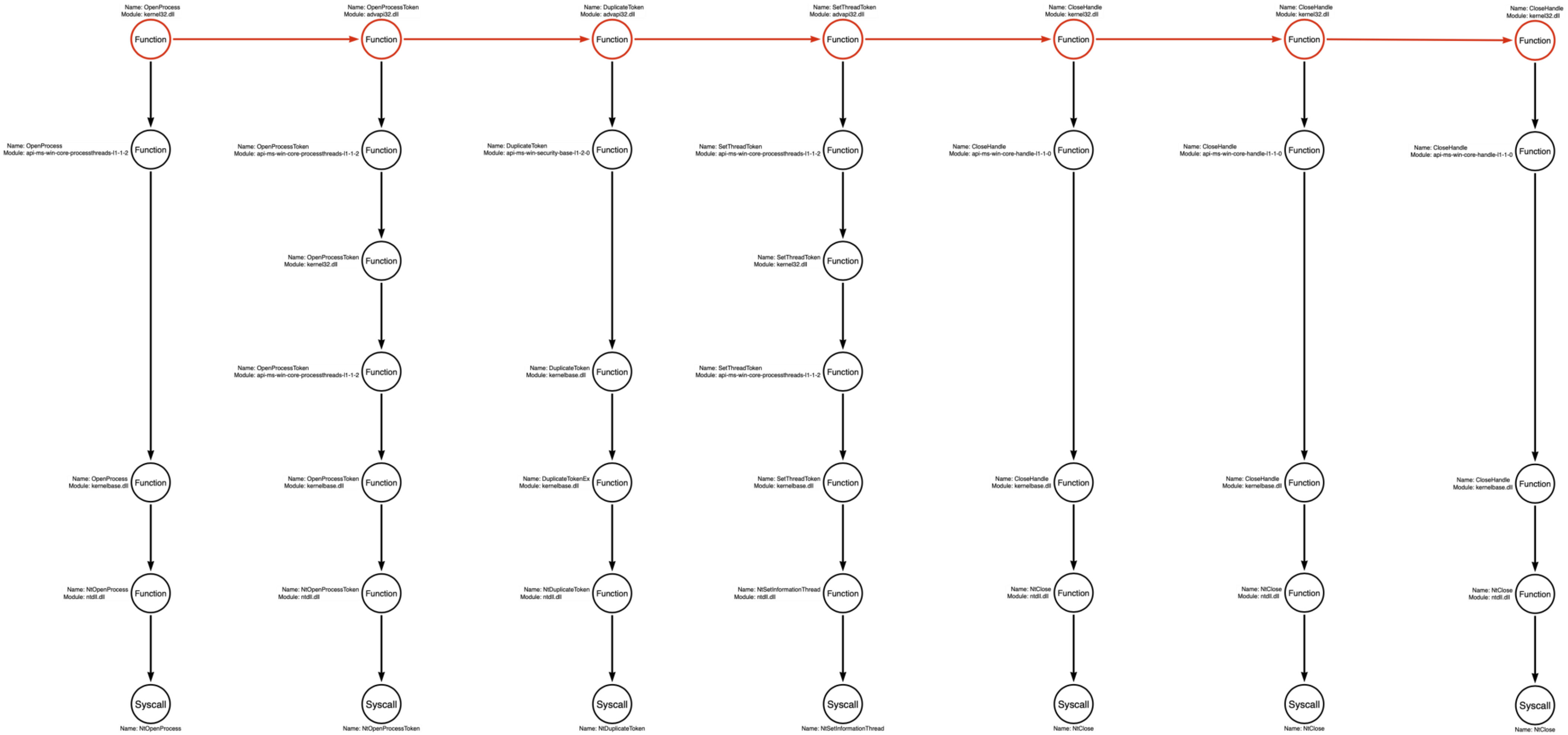
- SetThreadToken is an example of a classic simple function.
  - A simple function is a function whose call stack has no branches.
  - It does one thing and one thing only.
- This analysis allows us to connect the Win32 API function with its associated Native API call.
  - SetThreadToken -> NtSetInformationThread
- Did you look at the parameters passed to NtSetInformationThread?
  - ThreadInformationClass Parameter
    - ThreadImpersonationToken



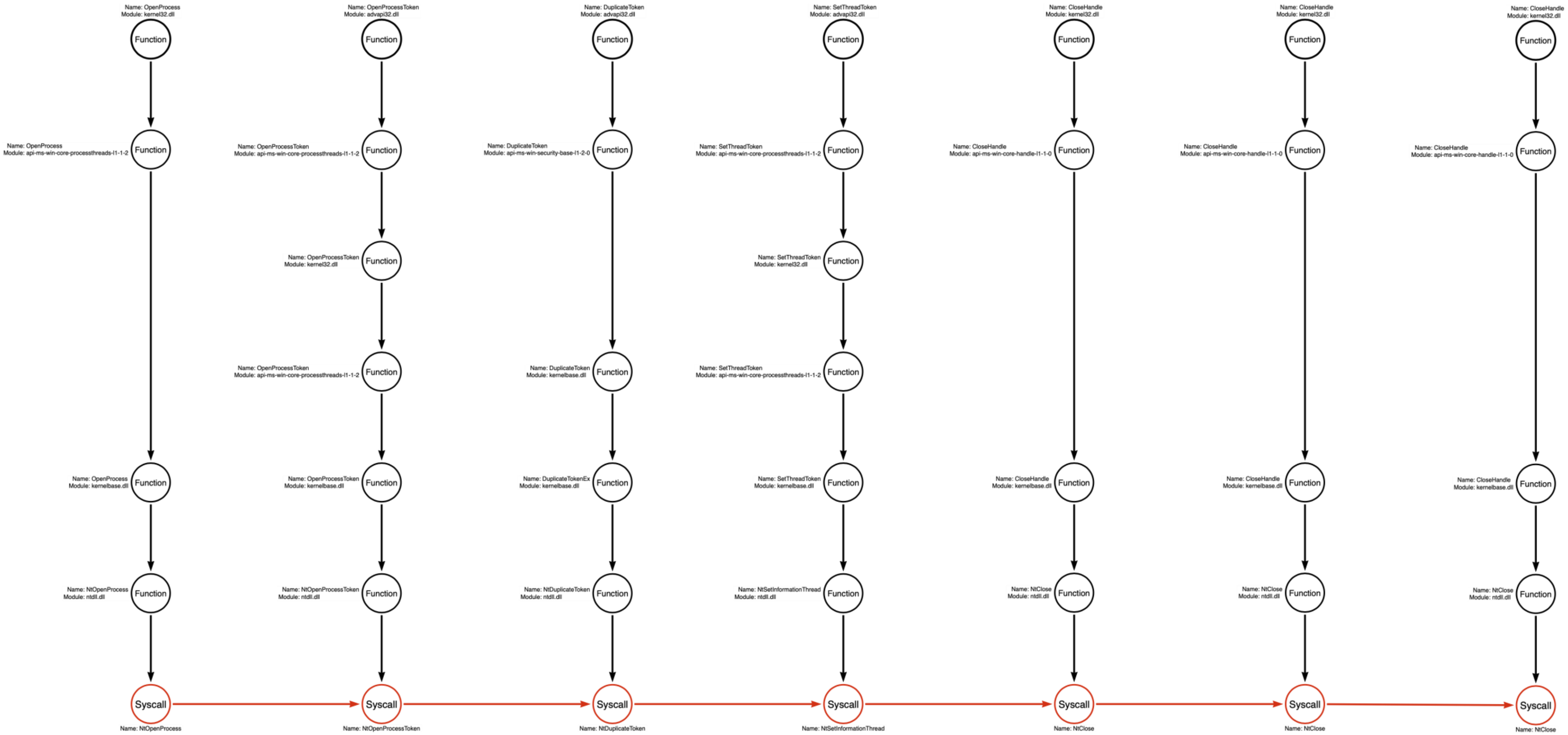
Function Call Stacks Based on Samples 1 & 2



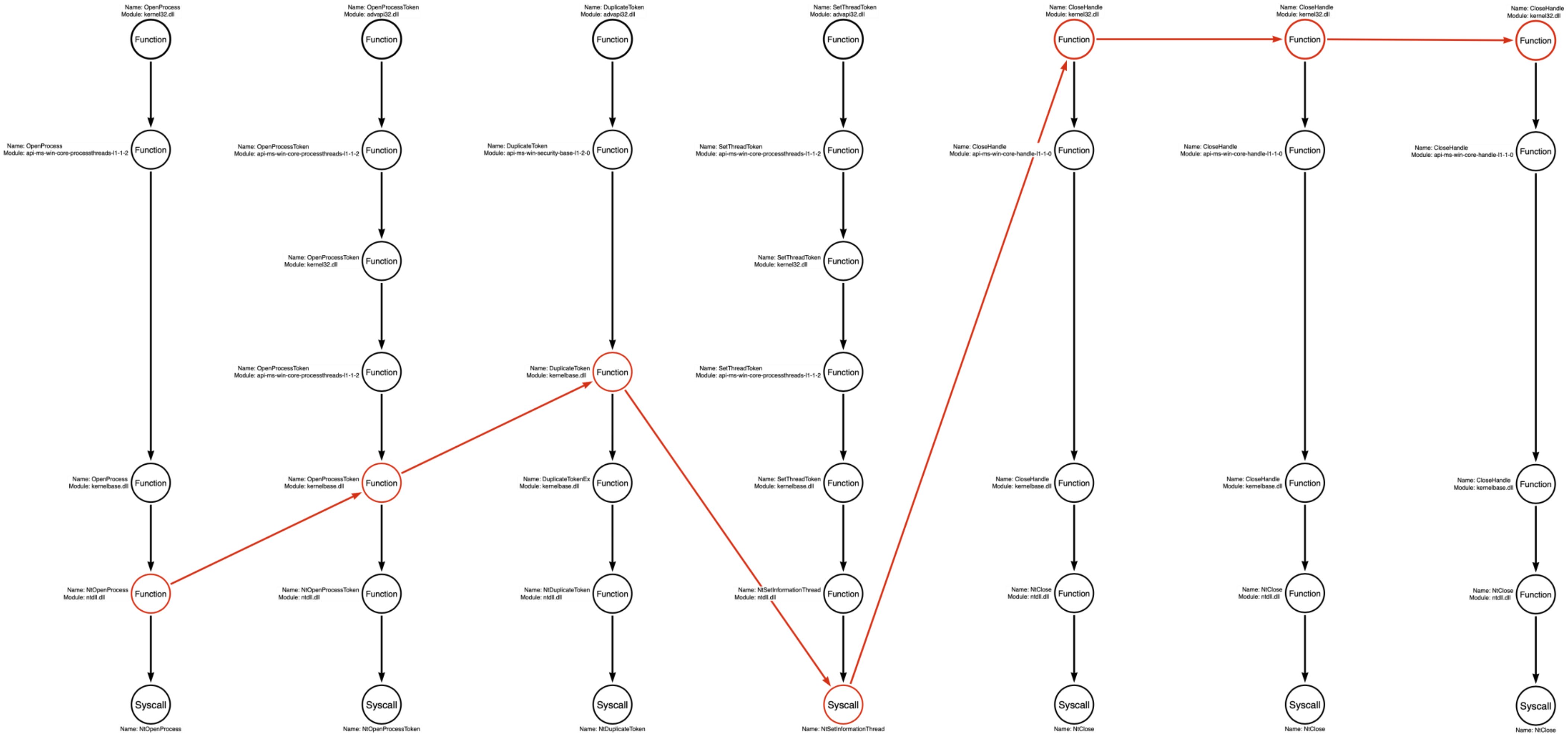
Real Variation from Samples 1 & 2



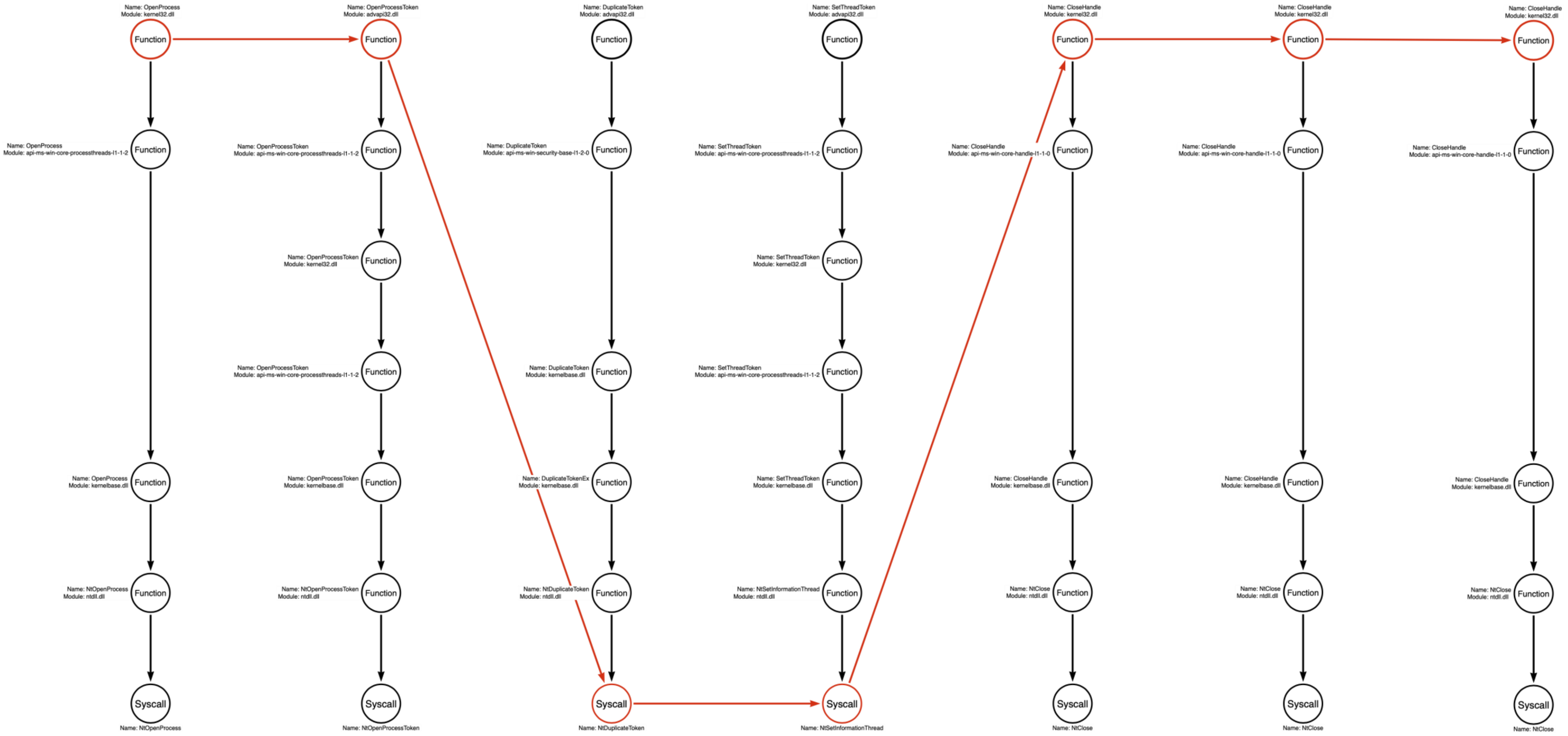
Hypothetical Variation #1



Hypothetical Variation #2



Hypothetical Variation #3

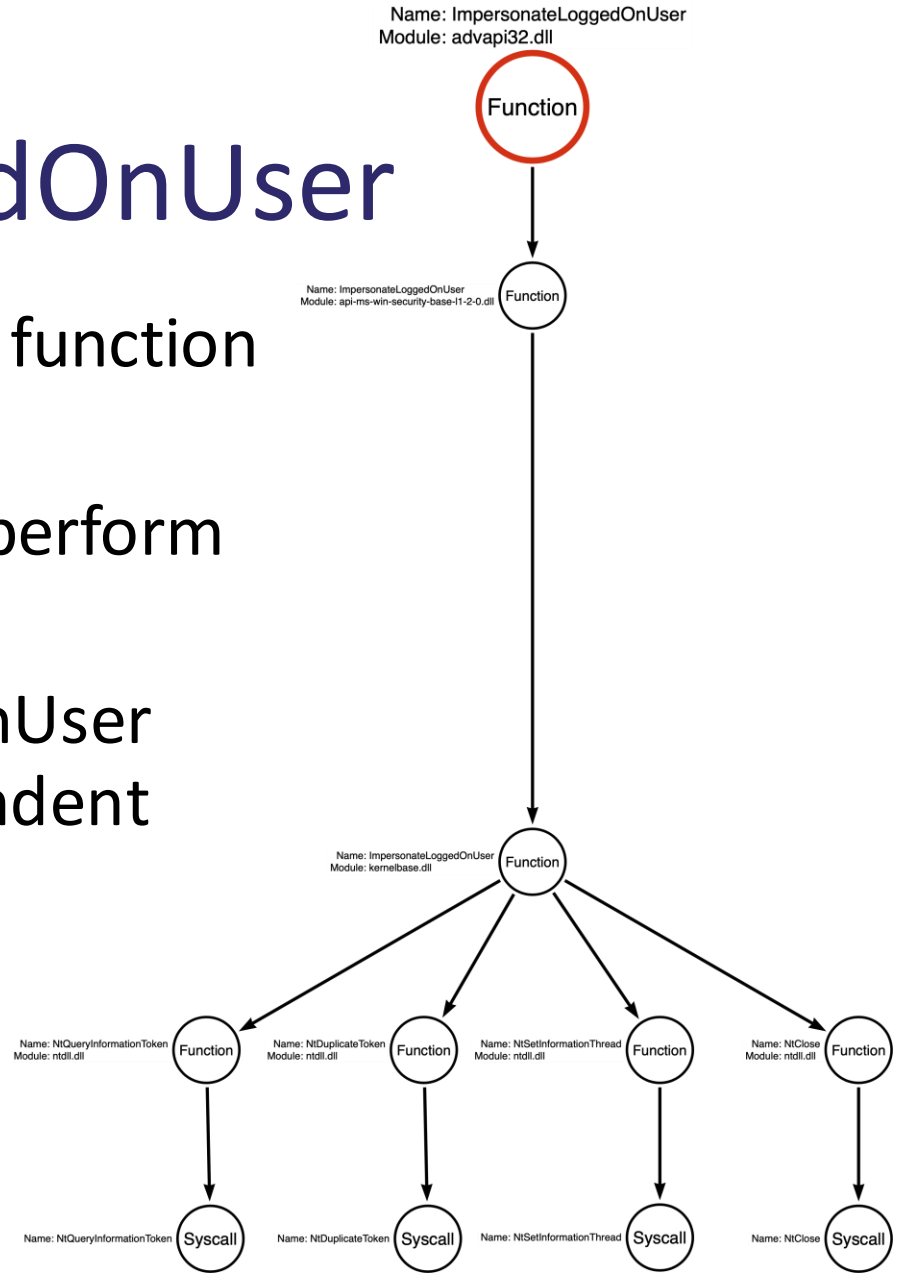


Lab 3: Analyzing ImpersonateLoggedOnUser

Analyzing the function call stack for the ImpersonateLoggedOnUser API function. This is an example of a "compound" function which is best conceived of as a miniature application that does some heavy lifting for the developer. We will see that this function does many things that previously required numerous function calls.

Call Stack - ImpersonateLoggedOnUser

- ImpersonateLoggedOnUser is a special type of function that we call a "compound function."
- Unlike simple functions, compound functions perform many actions via a single API call.
- In this case, we see that ImpersonateLoggedOnUser replaces the need to call four different independent functions.
  - NtQueryInformationToken
  - NtDuplicateToken (DuplicateToken)
  - NtSetInformationThread (SetThreadToken)
  - NtClose (CloseHandle)



What is an Operation

- An abstract category or group that contains interchangeable individual function calls.
  - This group is often represented by a function call stack, however, it is possible for multiple function call stacks to perform the same operation.
    - Ex. DuplicateToken and DuplicateTokenEx
- Defined as "an action taken against a securable object."
  - Ex 1. Process Create
  - Ex 2. Registry Key Read
  - Ex 3. File Delete
- Operations allow analysts to ignore the functional detail and focus on the behavior itself.
  - kernel32!OpenProcess vs. ntdll!NtOpenProcess
  - advapi32!SetThreadToken vs. advapi32!ImpersonateLoggedOnUser

Grammar Heuristic

- Behavioral Detection is based on an OBJECTIVE Point of View.
- Subject – Verb – Object
  - The process, named "beacon.exe," enumerated the subkeys of the Registry Key, named "HKLM\SYSTEM\CurrentControlSet\Services."
    - Process ENUMERATED Registry Key (Registry Key Enumerate)
  - The process, named "powershell.exe," created the file "C:\Windows\Temp\a.txt."
    - Process CREATED File (File Create)
- The Subjective POV allows for too many different interpretations and lacks the crucial context for filtering noise.
  - ~~Process CREATED File (Process Create)~~
  - ~~User CREATED File (User Create)~~
  - ~~Computer CREATED File (Computer Create)~~

Lab 4: Operations

In this lab, we will work through the process of creating an abstract category to contain the entire function call stack. This allows us to ignore the minor differences between two functions like kernel32!OpenProcess and kernelbase!OpenProcess and instead refer to the collection as the "Process Open" operation.

Open*/Create* Functions are Special

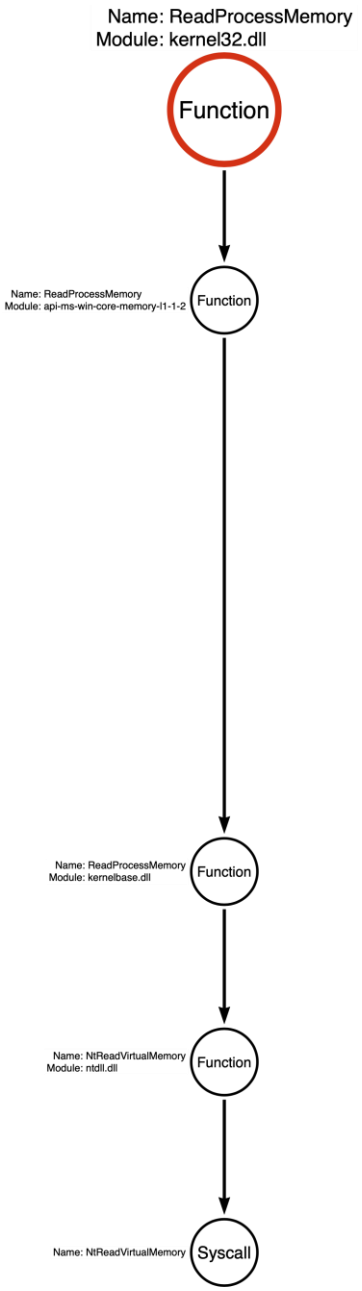
- Windows has several functions that facilitate opening a handle to securable objects.
  - OpenProcess, OpenService, OpenThread, OpenProcessToken, etc.
  - CreateFile, CreateService, CreateProcess
- Opens a handle to the object referenced.
- The caller must specify the types of actions it will take on the object.
  - Access Checks are performed when opening the handle.
- Handle is then passed to other functions to perform actions.
  - Process Ex: GetProcessInformation, ReadProcessMemory, TerminateProcess
  - Token Ex: GetTokenInformation, DuplicateToken

Determining the Target Object

- The target object of the function can be determined by following these simple steps:
  - If the function is prefixed by the term "Open" then the target object will be the object that is being opened.
    - OpenProcess = Process Open
  - If the function only takes one parameter of type HANDLE, then the HANDLE represents the target object.
  - If the function takes two or more parameters of type HANDLE, but only one is an input parameter then the input parameter represents the target object.
  - If the function takes two or more parameters of type HANDLE and more than one are input parameters, then analyze the Native API at the bottom of the call stack to identify the target object.

Ex 1: ReadProcessMemory

- Only one parameter is of the type HANDLE.
- In this case it is a handle to a process object.
- This indicates that the object of this action is a Process object.



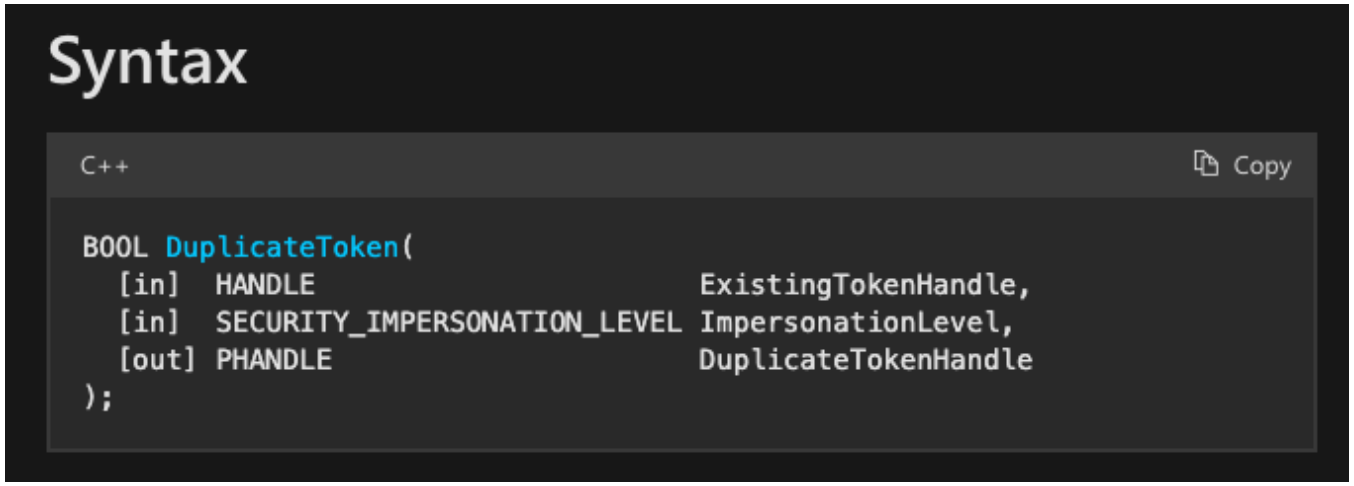
Syntax

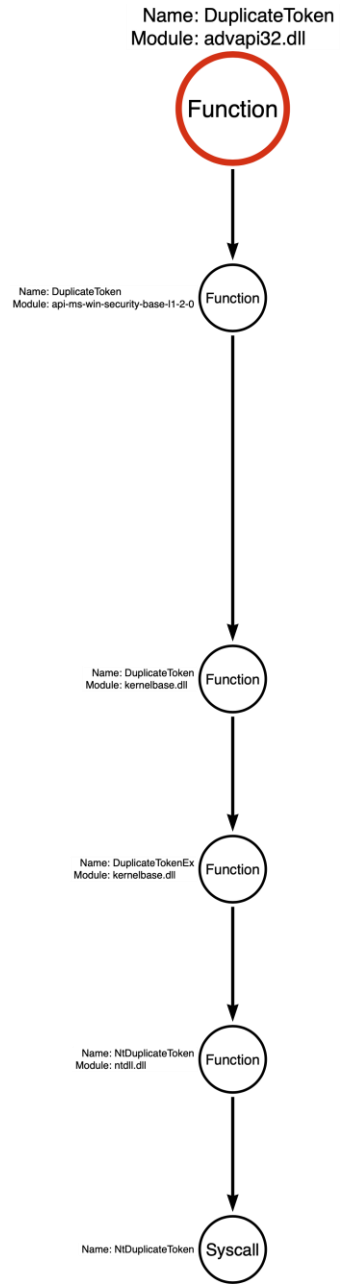
C++ Copy

```
BOOL ReadProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPCVOID lpBaseAddress,
  [out] LPVOID  lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesRead
);
```

Ex 2: DuplicateToken

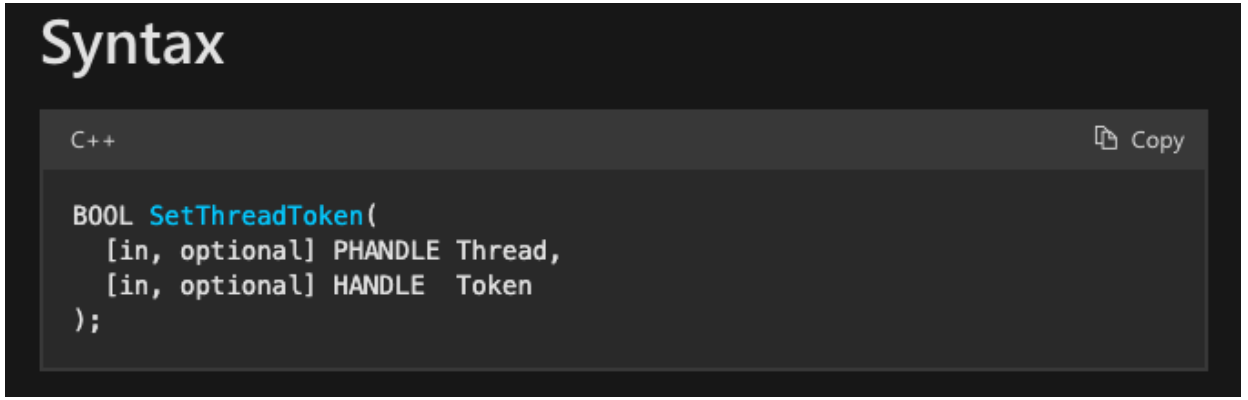
- DuplicateToken is a function that has two HANDLE parameters.
- Both handles refer to Token objects.
- One handle is an input parameter (ExistingTokenHandle) while the other is an output parameter (DuplicateTokenHandle).
- The input parameter, a TOKEN handle, is the target object.

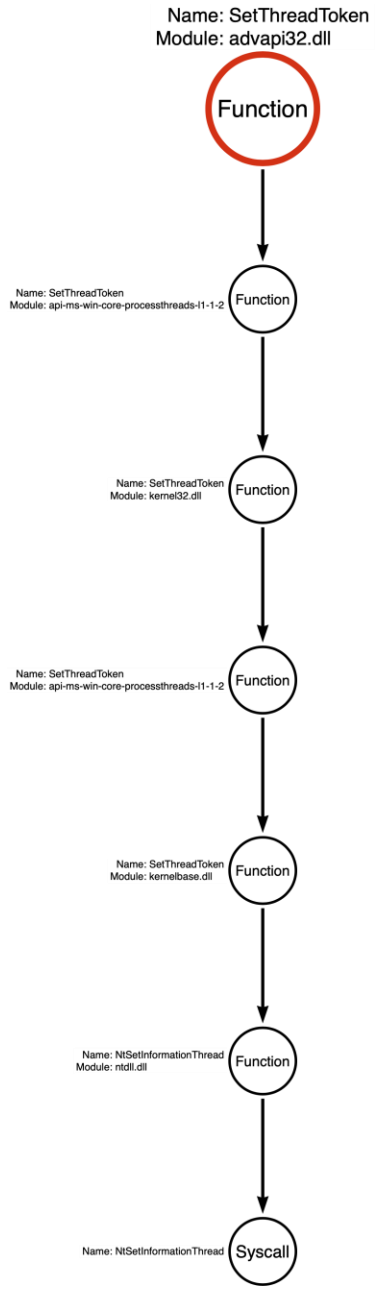




Ex 3: SetThreadToken

- This example has two parameters of type HANDLE.
  - Thread and Token
- Both handles are specified as input parameters.
- The Native API at the bottom of the stack is NtSetInformationThread.
- The Thread handle is the target object.

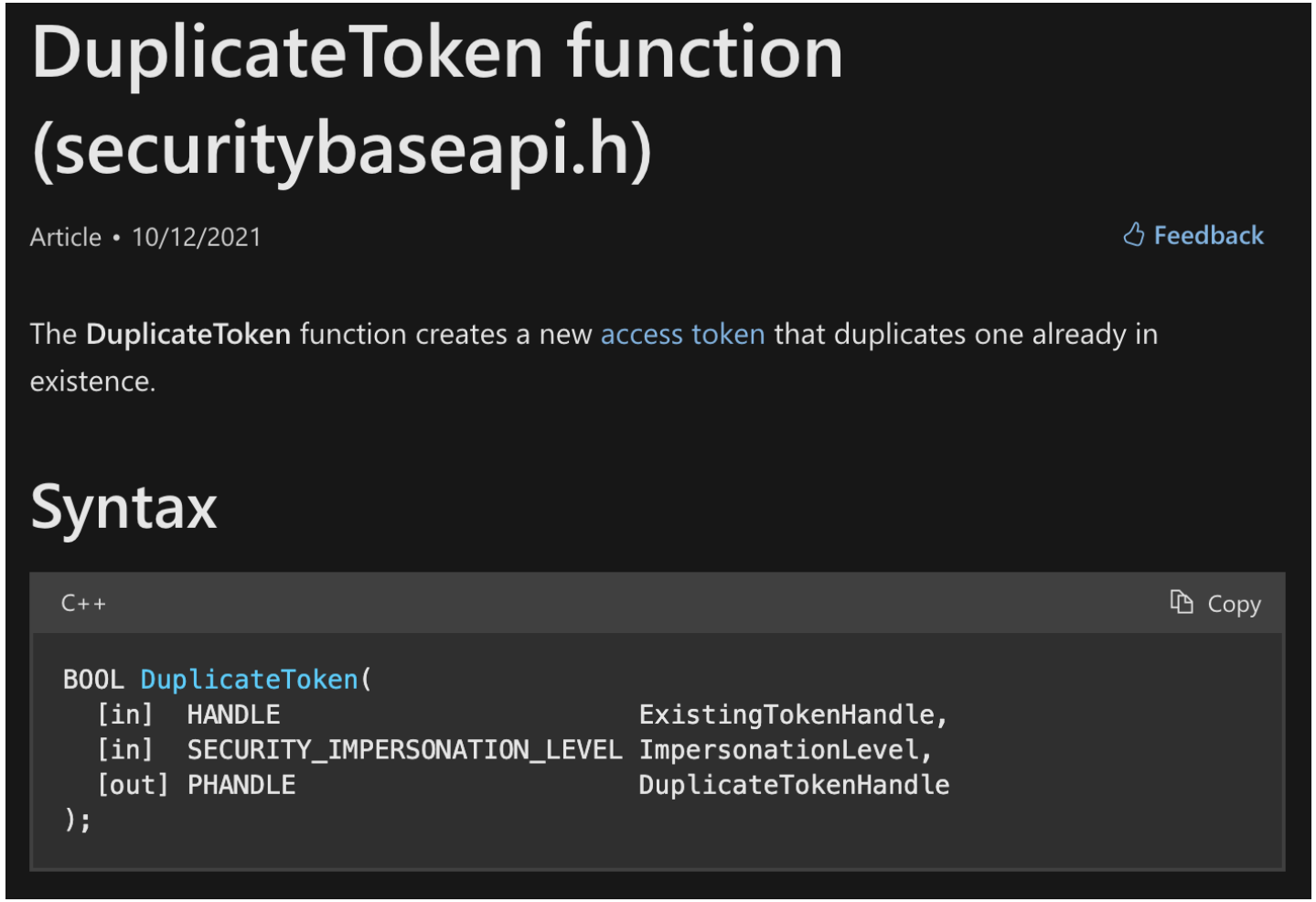




Determining the Action

1. Visit the documentation for the function in question.
2. Find the description for the object's HANDLE type parameter.
3. Identify any reference to required access or access rights."
   - A handle to a _____ opened with _____ access."
4. Google "<OBJECT> Security and Access Rights" and visit the page.
5. Find the referenced access rights.
   - If more than one access right is required, then it is probably a compound function.
6. Match your action as closely as possible to the access right's name.
   - TOKEN_DUPLICATE -> Duplicate

Step 1: Visit Function Documentation



DuplicateToken function (securitybaseapi.h)

Article • 10/12/2021 Feedback

The **DuplicateToken** function creates a new access token that duplicates one already in existence.

Syntax

C++ Copy

```
BOOL DuplicateToken(
  [in]  HANDLE                       ExistingTokenHandle,
  [in]  SECURITY_IMPERSONATION_LEVEL ImpersonationLevel,
  [out] PHANDLE                      DuplicateTokenHandle
);
```

Step 2: Find the parameter description

Syntax

C++ Copy

```
BOOL DuplicateToken(
  [in]  HANDLE                        ExistingTokenHandle,
  [in]  SECURITY_IMPERSONATION_LEVEL  ImpersonationLevel,
  [out] PHANDLE                       DuplicateTokenHandle
);
```

Parameters

`[in] ExistingTokenHandle`

A handle to an access token opened with TOKEN_DUPLICATE access.

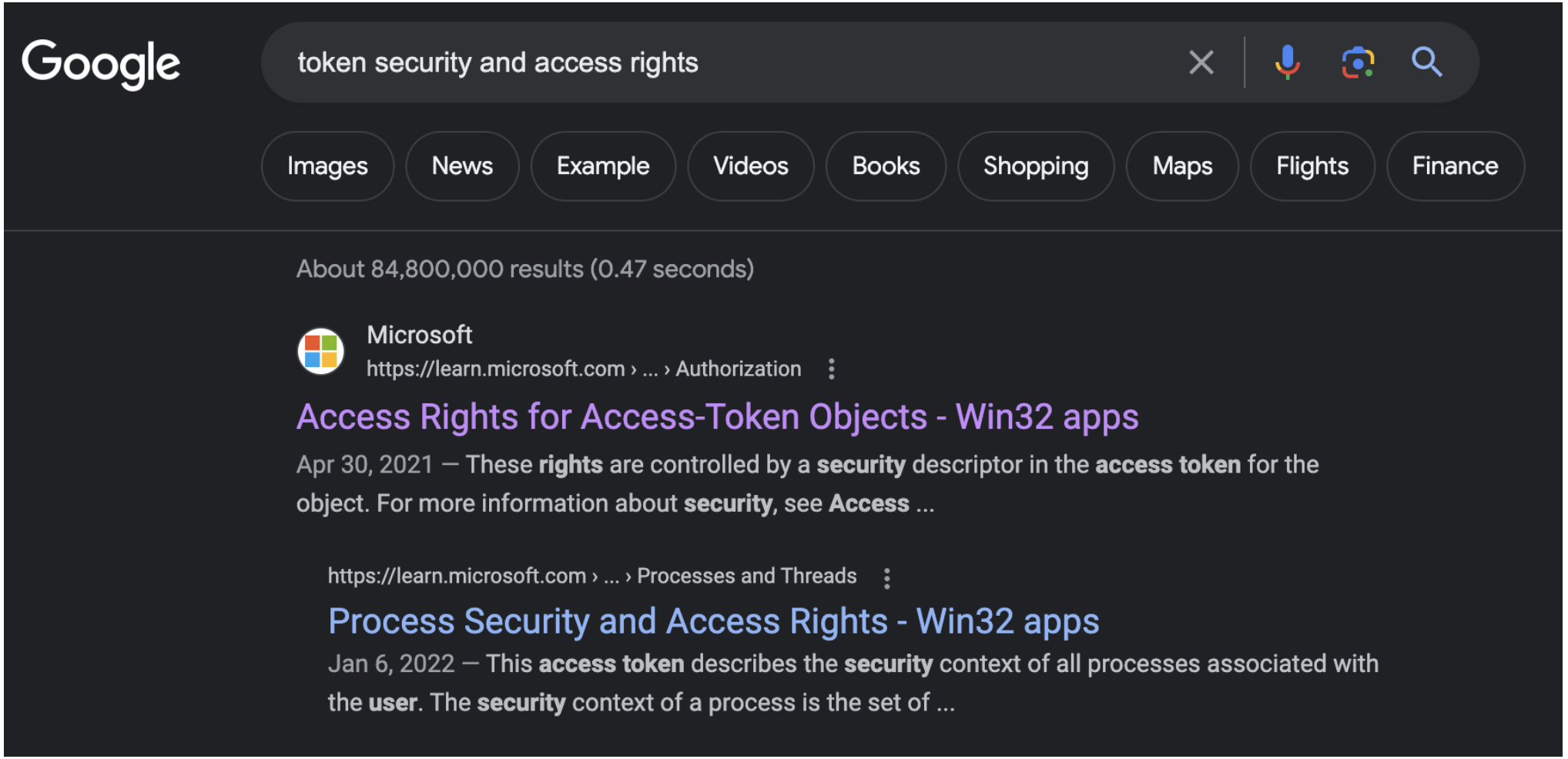
Step 3: Identify required Access Right(s)

`[in] ExistingTokenHandle`

A handle to an access token opened with TOKEN_DUPLICATE access.

https://learn.microsoft.com/en-us/windows/win32/secauthz/access-rights-for-access-token-objects

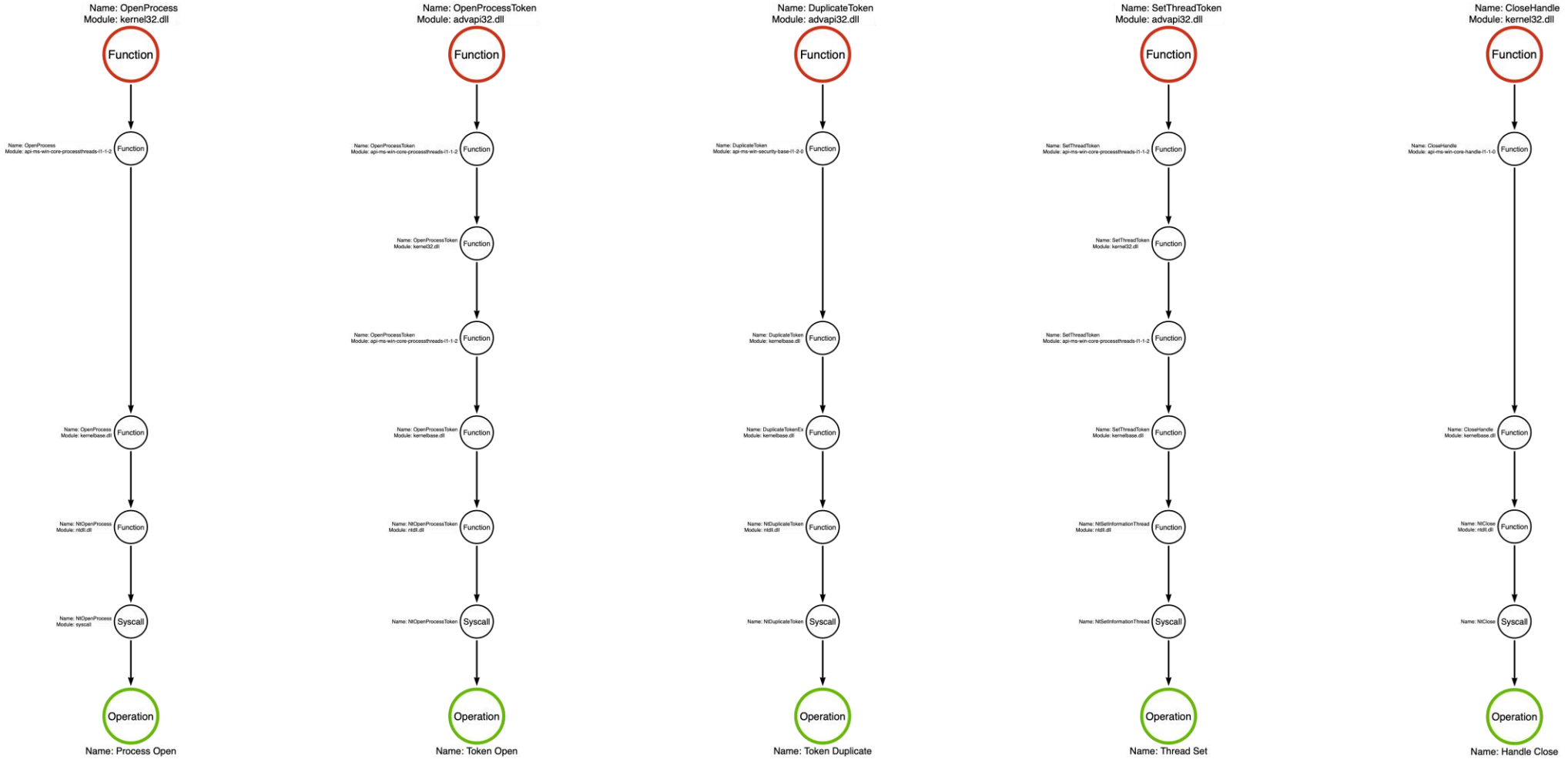
Step 4: Google "Token Security..."



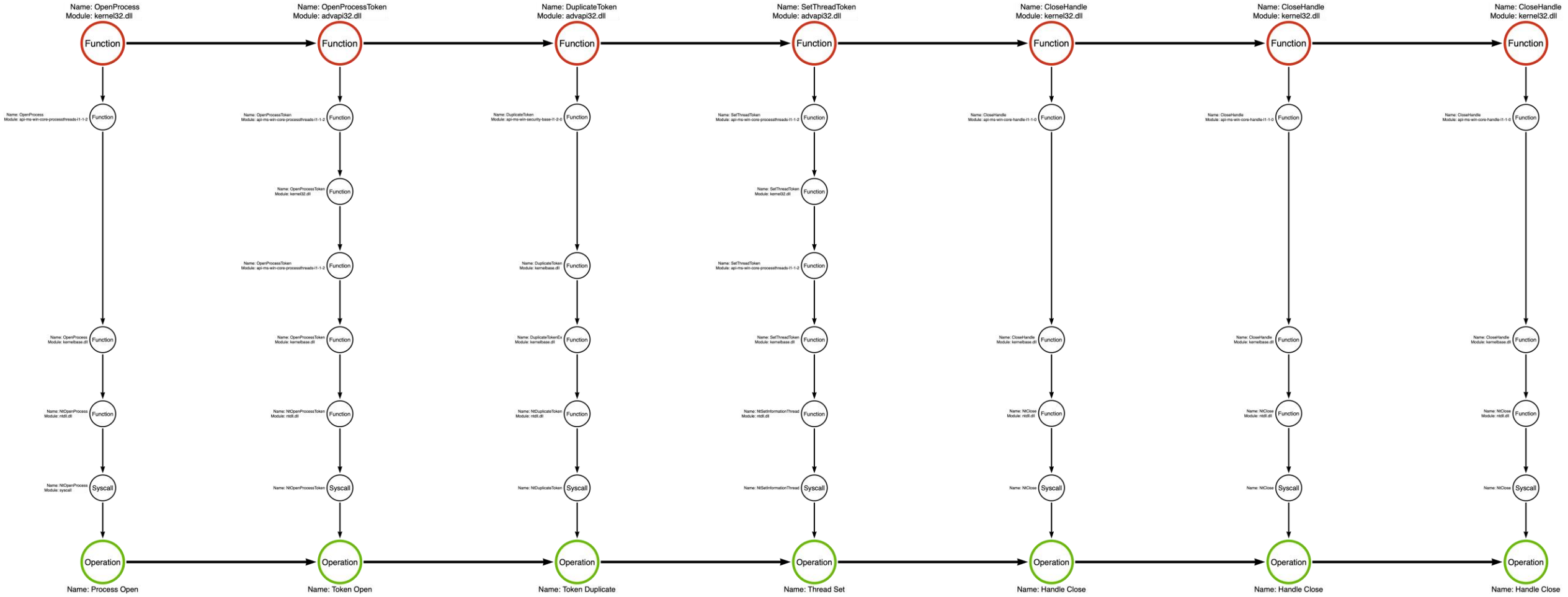
Step 5: Find referenced Access Right

Value	Meaning
TOKEN_ADJUST_DEFAULT	Required to change the default owner, primary group, or DACL of an access token.
TOKEN_ADJUST_GROUPS	Required to adjust the attributes of the groups in an access token.
TOKEN_ADJUST_PRIVILEGES	Required to enable or disable the privileges in an access token.
TOKEN_ADJUST_SESSIONID	Required to adjust the session ID of an access token. The SE_TCB_NAME privilege is required.
TOKEN_ASSIGN_PRIMARY	Required to attach a *primary token* to a *process*. The SE_ASSIGNPRIMARYTOKEN_NAME privilege is also required to accomplish this task.
TOKEN_DUPLICATE	Required to duplicate an access token.
TOKEN_EXECUTE	Same as STANDARD_RIGHTS_EXECUTE.
TOKEN_IMPERSONATE	Required to attach an impersonation access token to a process.

Sample 1: Function Call Stacks with Operations



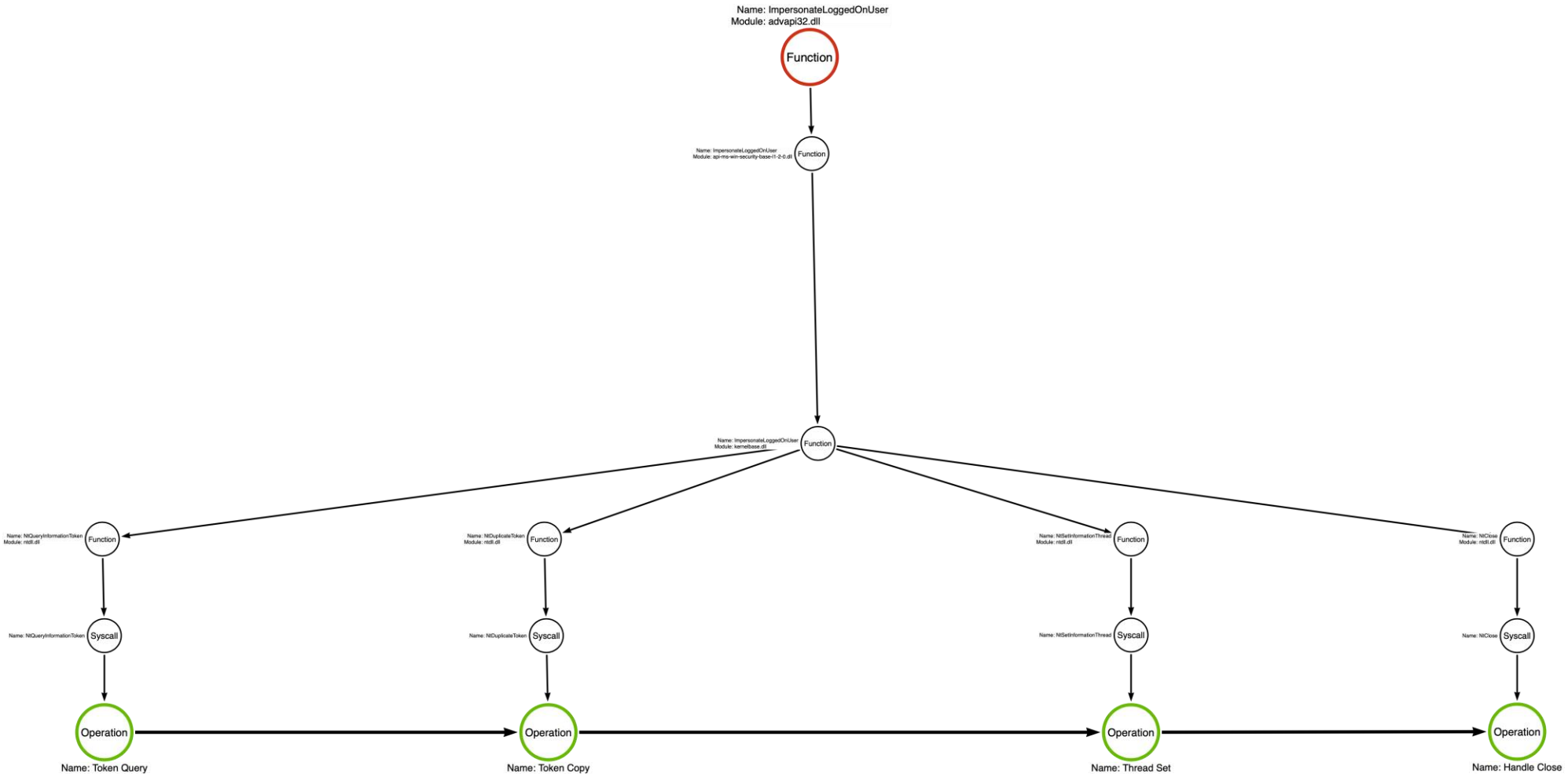
Tool Graph: Sample 1 and 2



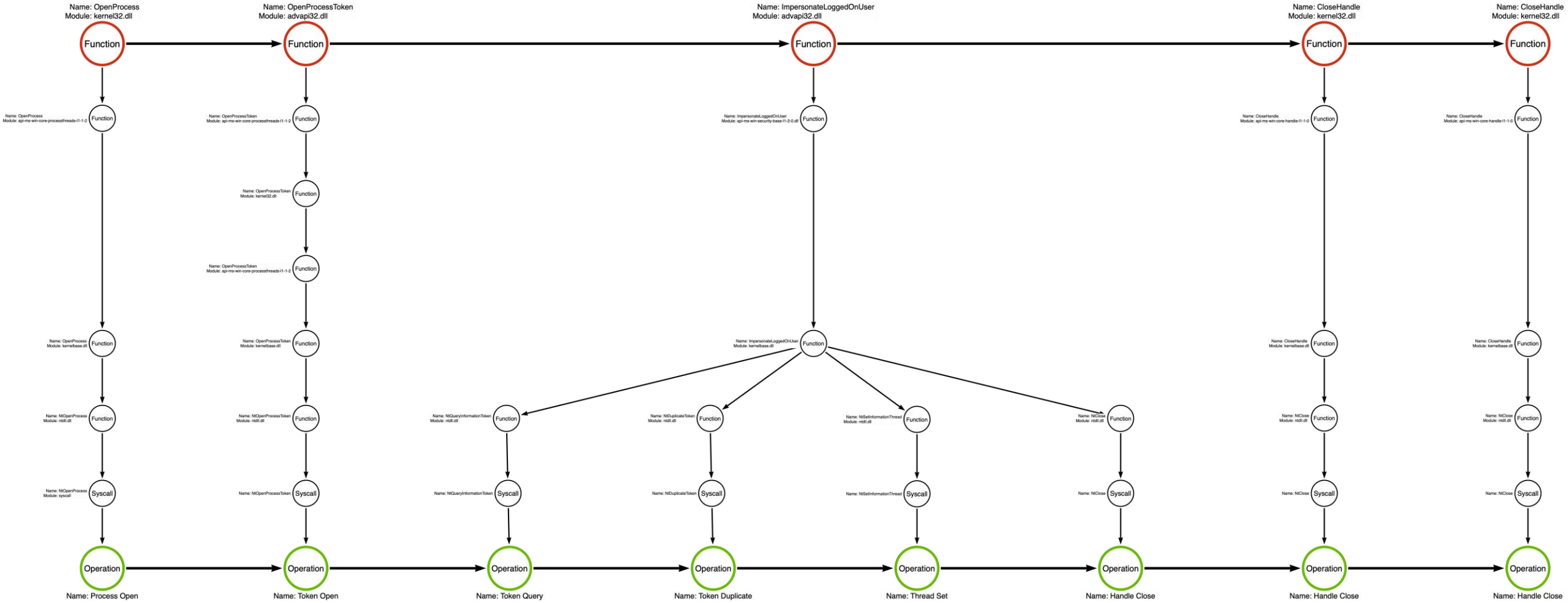
Operation Chain – The next rank

- Samples 1 & 2 demonstrated only one possible functional variation.
- Function call stacks expose additional possible variations.
  - OpenProcess – 5 functions
  - OpenProcessToken – 7 functions
  - DuplicateToken – 6 functions
  - SetThreadToken – 7 functions
  - CloseHandle x3 – 5 functions
- We can calculate functional variations represented by operation chain by multiplying the number of functions in each stack.
  - 5x7x6x7x5x5x5 = 183,750 functional variations

Sample 2: ImpersonateLoggedOnUser

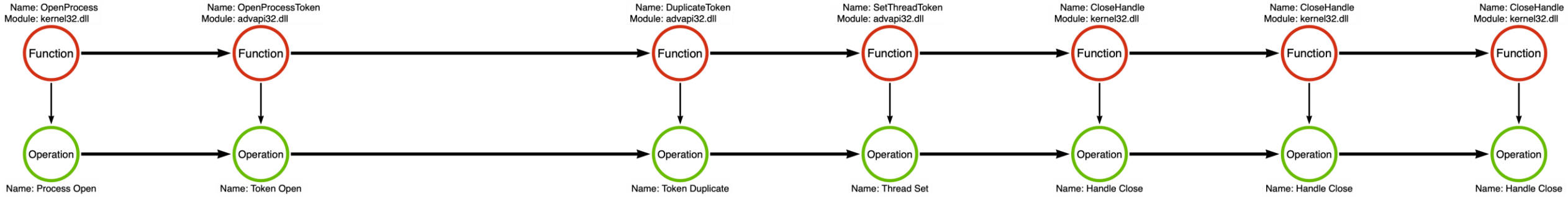


Tool Graph: Sample 3

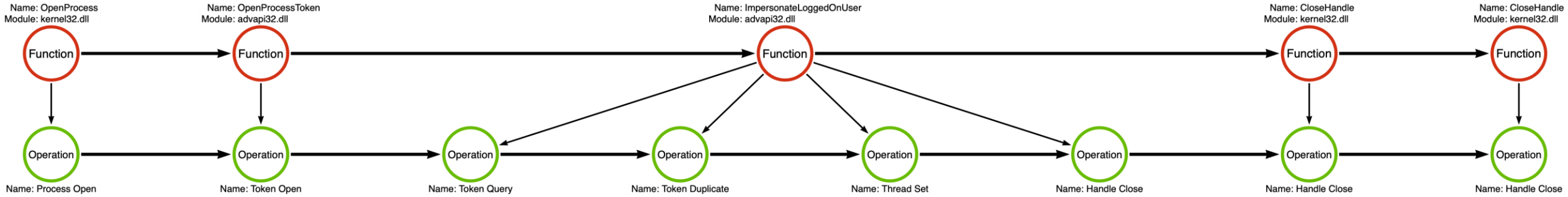


Function Chain vs. Operation Chain

Sample 1 and 2

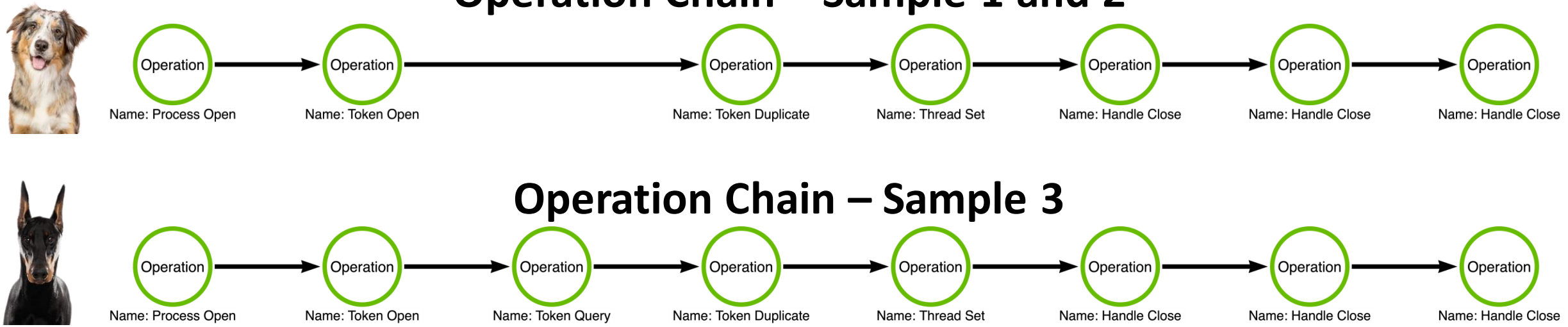


Sample 3



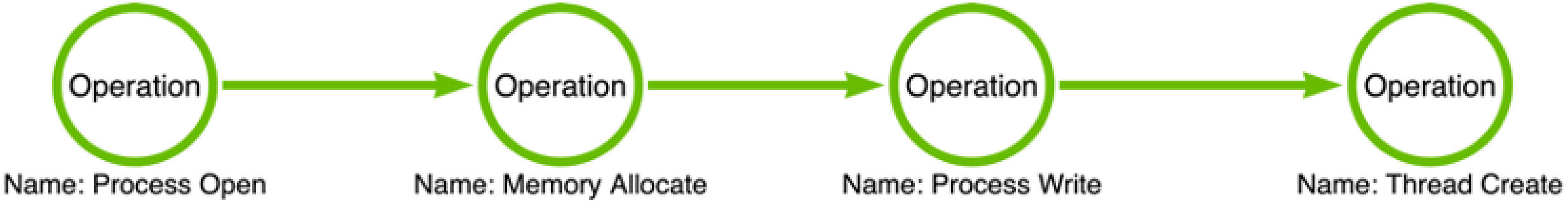
Operation Chain Comparison

- Through an operational lens, the two samples appear equivalent.
- Think of these as belonging to different species, but the same genus.
  - SetThreadToken is Austrailian Shepherd, ImpersonateLoggedOnUser is Doberman.
- This means that an operationally focused analytic may detect both.

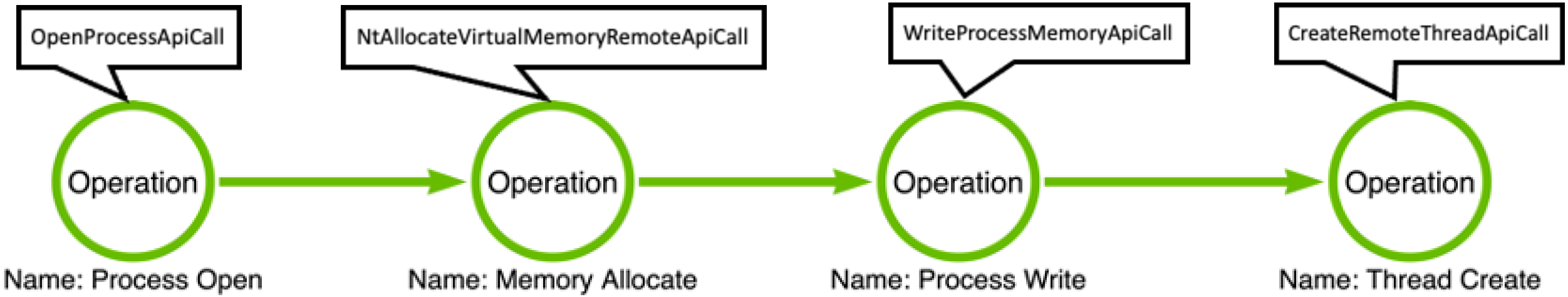


We See Operations

Sample Operation Chain – Process Injection



Operation Chain + Events (MDE Edition)



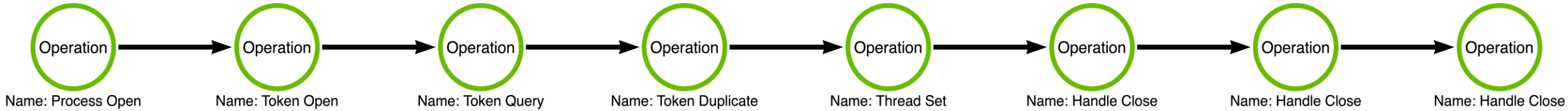
Implicit Process Create

- You've probably seen the graph showing that Process command-line events are the most common event for ATT&CK Techniques.
- You probably know that Process Creation events are the most common foundation of detection rules.
- How then is that the case, if there is no Process Create operation in this (or most other) operation chains?
  - If we perceive operations?

Operation Chain Simplification

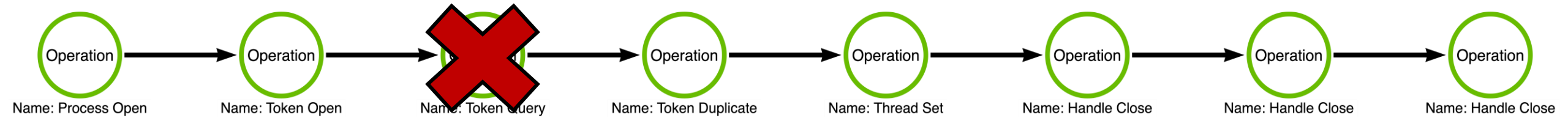
Simplification of Sample 3

- Once an operation chain or a few operation chains have been established, it is useful to simplify the chains.
  - Simplification is essentially the removal of unnecessary operation from the chain.
- Simplification offers many benefits:
  - Reduce combinatorial complexity.
  - Eliminates the threat of "junk operations."
  - Helps to focus on operations that are most valuable for detective or preventative controls.



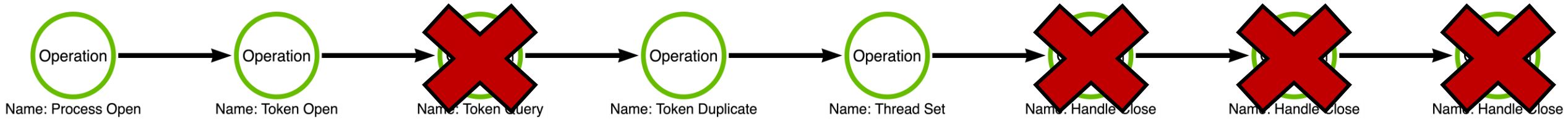
Simplification of Sample 3 – Token Query

- It may not be immediately obvious that the Token Query operation is unnecessary.
- The easy answer is that because it is the only difference between Sample 1/2's and Sample 3's operation chains, we know it can be excluded.
  - It turns out that ImpersonateLoggedOnUser uses it to determine if the provided Token is a Primary or Impersonation Token.
  - In the case of SetThreadToken, the caller would have opened the Token Handle explicitly, so they would likely know the Token Type.



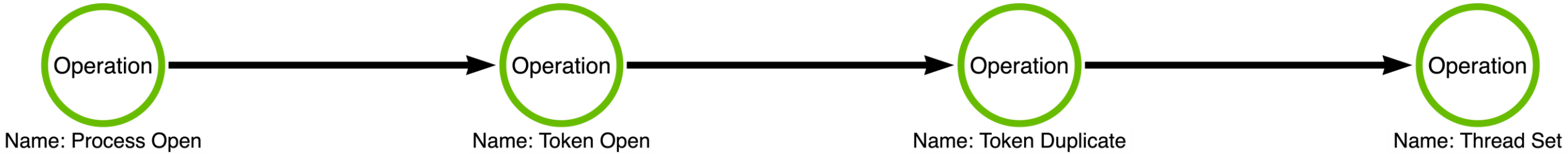
Simplification of Sample 3 – Handle Close

- This is a decision that was alluded to earlier when discussing the number of functional variations represented by the tool graph.
- Closing handles is something that is not MANDATORY for developers to do to successfully Impersonate.
  - It is considered a good programming practice.
  - Even legitimate applications sometimes have issues with handle leaks.
- Since it is not mandatory and offers very little contextual detail, we can remove it from our simplified operation chain.

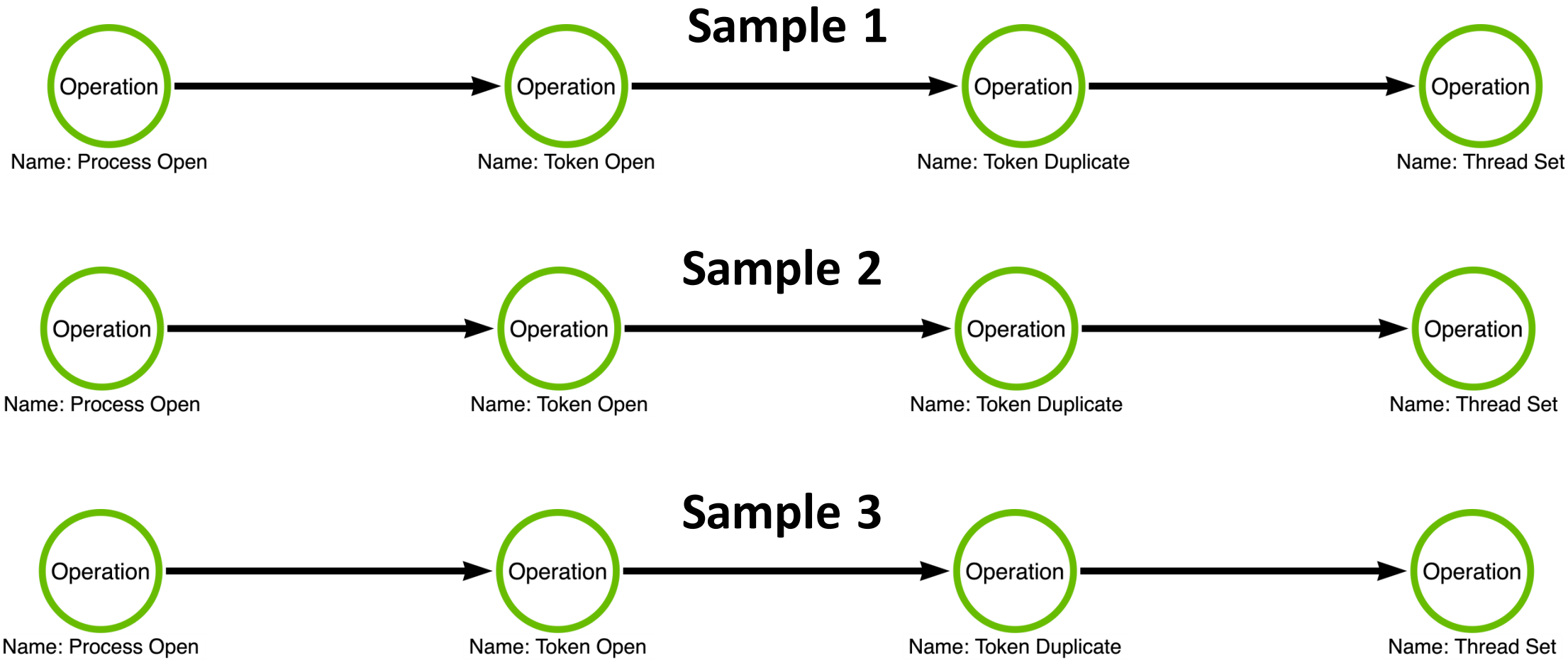


Sample 3 – Reduced

- Once unnecessary operations have been identified they can be removed from the chain.
  - This allows analysts to ignore insignificant differences between samples such as "forgetting" to close a handle.
- The simplified operation chain can then be used to evaluate the similarity of two functionally different, but operationally similar samples.
  - For instance, what happens when we compare Samples 1 & 2 with Sample 3?



Comparison of Simplified Chains – Samples 1, 2, & 3



Sample 4

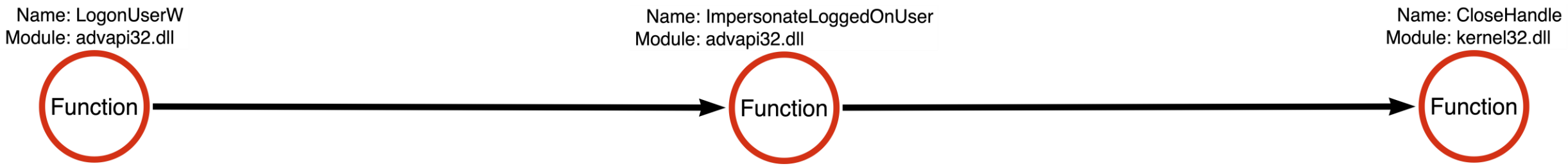
Sample 4: Source Code

```
16      if (GivenUser.find(delimiter) == std::string::npos) {
17          targetdomainname = L".";
18          targetusername = GivenUser;
19      }
20      else {
21          targetdomainname = GivenUser.substr(0, GivenUser.find(delimiter));
22          targetusername = GivenUser.substr(GivenUser.find(delimiter) + 1, GivenUser.length());
23      }
24
25      LogonUser(targetusername.c_str(), targetdomainname.c_str(), UserPassword.c_str(), LOGON32_LOGON_INTERACTIVE, LOGON32_PROVIDER_DEFAULT, &hToken);
26      ImpersonateLoggedOnUser(hToken);
27
28      CloseHandle(hToken);
```


https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%204/src/Source.cpp

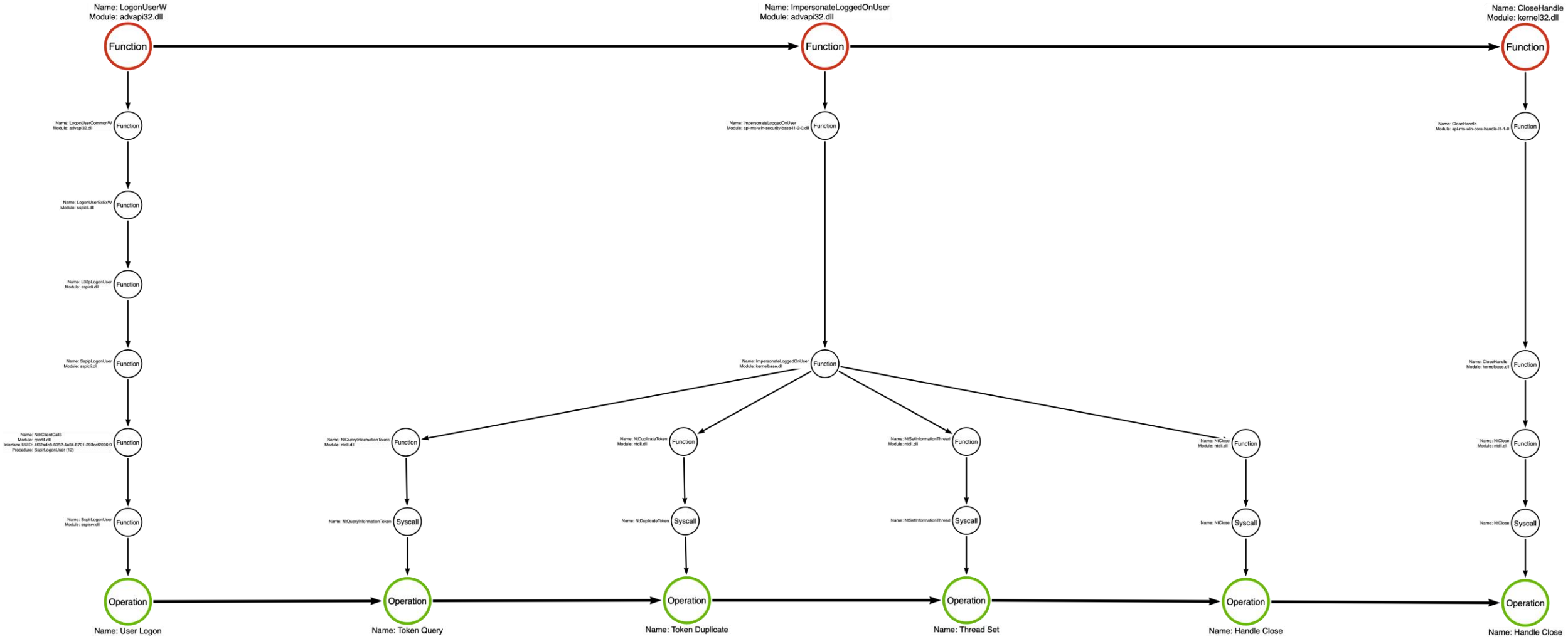
Sample 4: Source Code

```
16      if (GivenUser.find(delimiter) == std::string::npos) {
17          targetdomainname = L".";
18          targetusername = GivenUser;
19      }
20      else {
21          targetdomainname = GivenUser.substr(0, GivenUser.find(delimiter));
22          targetusername = GivenUser.substr(GivenUser.find(delimiter) + 1, GivenUser.length());
23      }
24
25      LogonUser(targetusername.c_str(), targetdomainname.c_str(), UserPassword.c_str(), LOGON32_LOGON_INTERACTIVE, LOGON32_PROVIDER_DEFAULT, &hToken);
26      ImpersonateLoggedOnUser(hToken);
27
28      CloseHandle(hToken);
```



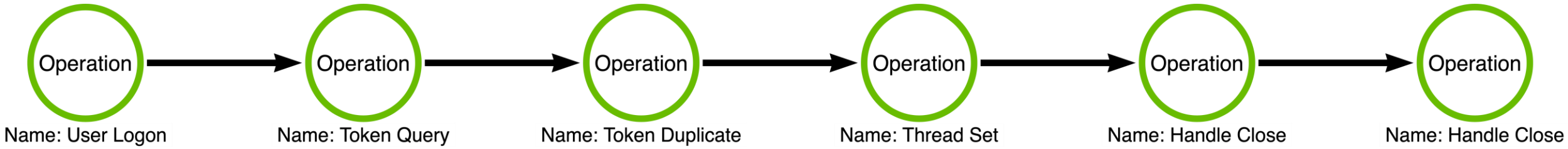
https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%204/src/Source.cpp

Tool Graph: Sample 4



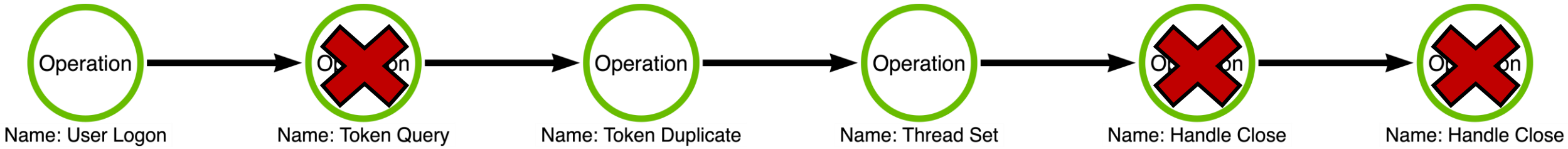
Operation Chain

- After analyzing the operation chain, we see that the middle portion of the chain bears a strong resemblance to the chains of the first three samples.
  - Token Duplicate -> Thread Set -> Handle Close -> Handle Close
- However, the first half of the chain is unique.
  - The User Logon operation is used to create a logon session for a user that was not previously logged on to the system.
  - As a result, an Access Token is returned which can be impersonated.



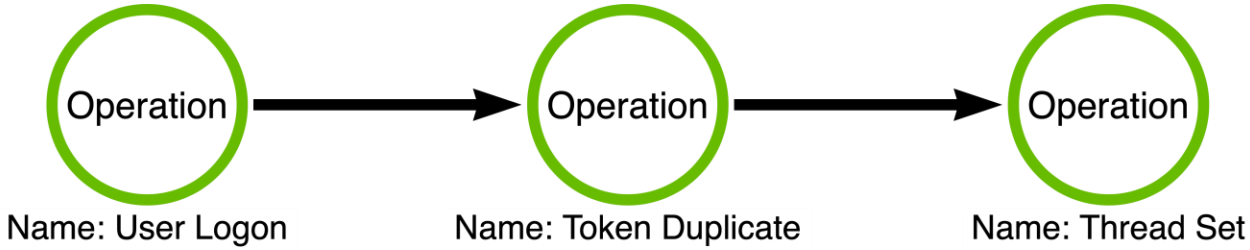
Simplification

- The following operations can be dropped as they are not necessary to make the operation chain valid:
  - Token Query
    - This is an artifact of the ImpersonateLoggedOnUser call, like Sample 3, and the same assessment applies that the same chain could be produced without it via SetThreadToken.
  - Handle Close
    - This closes the handle of the duplicated token after it has been set to the thread.
  - Handle Close
    - This closes the handle to the original token which was produced via the LogonUser call.



Morphological Analysis – Canis Lupus

- Grey Wolf
  - Family: Canidae
  - Genus: Canis
  - Species: Canis Lupus
- The change to how the handle of the target token is significant.
  - User Logon instead of Process Open -> Token Open.
- However, the "business end" remains constant.



Sample 5

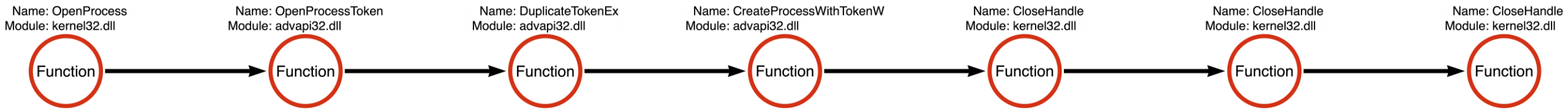
Sample 5: Source Code

```
10      HANDLE hToken, hDuplicate,hProcess = NULL;
11
12      STARTUPINFO si = { sizeof(si) };
13      PROCESS_INFORMATION pi;
14
15      hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, FALSE, PID);
16      OpenProcessToken(hProcess, TOKEN_QUERY | TOKEN_DUPLICATE, &hToken);
17      DuplicateTokenEx(hToken, TOKEN_QUERY | TOKEN_DUPLICATE | TOKEN_ASSIGN_PRIMARY | TOKEN_ADJUST_DEFAULT | TOKEN_ADJUST_SESSIONID, NULL, SecurityImpersonation, TokenPrimary, &hDuplicate);
18      CreateProcessWithTokenW(hDuplicate, LOGON_WITH_PROFILE, L"C:\\Windows\\System32\\cmd.exe", NULL, CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
19
20      CloseHandle(hDuplicate);
21      CloseHandle(hToken);
22      CloseHandle(hProcess);
```

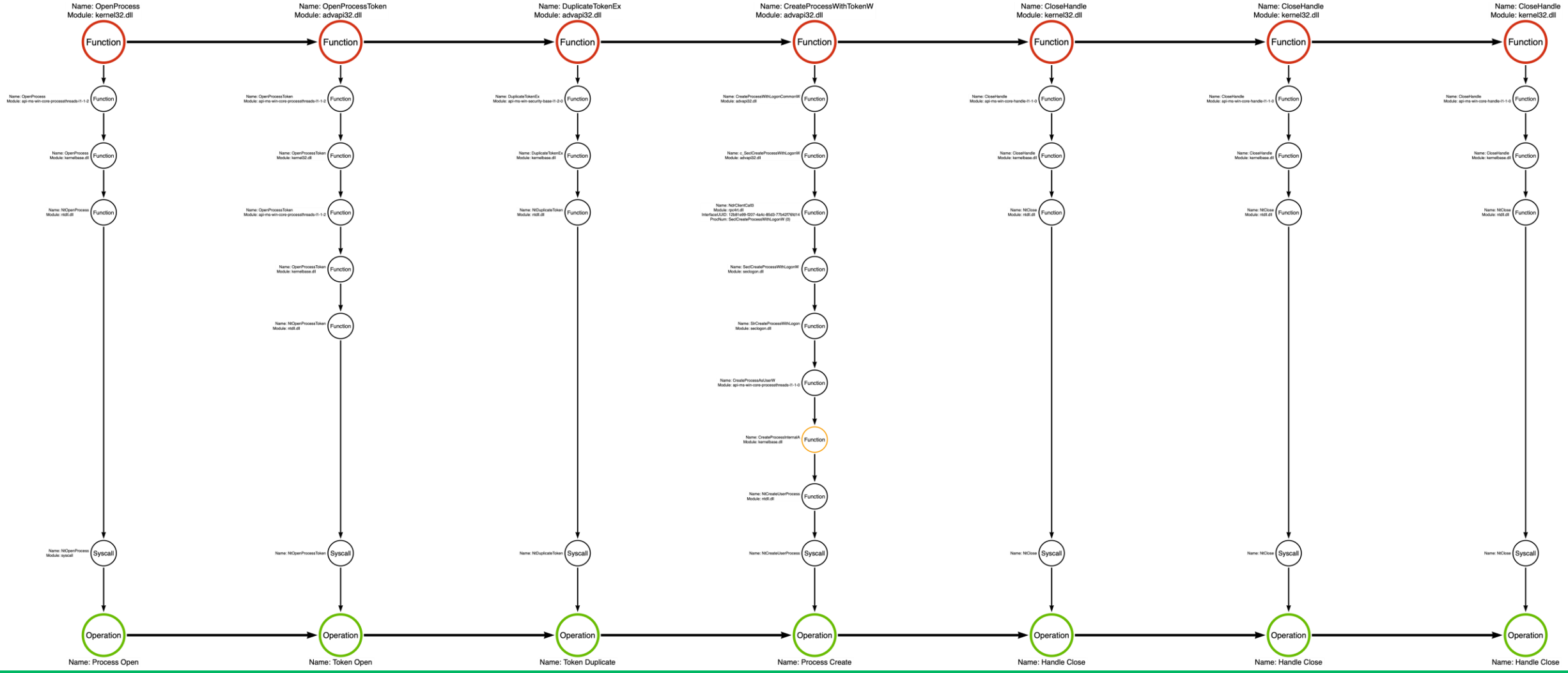

https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%205/src/Source.cpp

Sample 5: Source Code

```
10      HANDLE hToken, hDuplicate,hProcess = NULL;
11
12      STARTUPINFO si = { sizeof(si) };
13      PROCESS_INFORMATION pi;
14
15      hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, FALSE, PID);
16      OpenProcessToken(hProcess, TOKEN_QUERY | TOKEN_DUPLICATE, &hToken);
17      DuplicateTokenEx(hToken, TOKEN_QUERY | TOKEN_DUPLICATE | TOKEN_ASSIGN_PRIMARY | TOKEN_ADJUST_DEFAULT | TOKEN_ADJUST_SESSIONID, NULL, SecurityImpersonation, TokenPrimary, &hDuplicate);
18      CreateProcessWithTokenW(hDuplicate, LOGON_WITH_PROFILE, L"C:\\Windows\\System32\\cmd.exe", NULL, CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
19
20      CloseHandle(hDuplicate);
21      CloseHandle(hToken);
22      CloseHandle(hProcess);
```

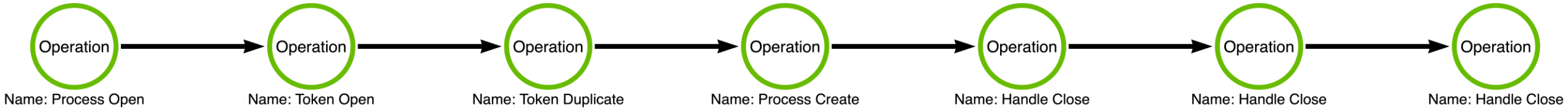


Tool Graph: Sample 5



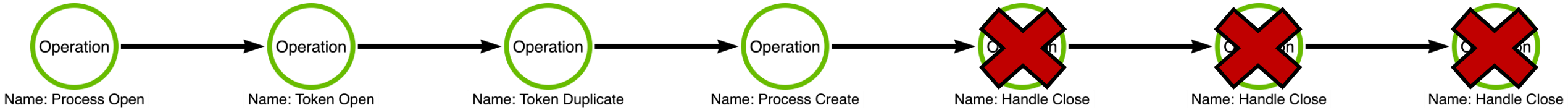
Operation Chain

- Sample 5 is similar to Samples 1 & 2 with one key difference.
  - The Thread Set operation seen in the first two samples is replaced with a Process Create operation in this sample.
  - This change is relatively profound in the sense that the impersonation is being applied to a new process instead of the existing process.
  - Questions might arise surrounding the impact of this change on "detectability."



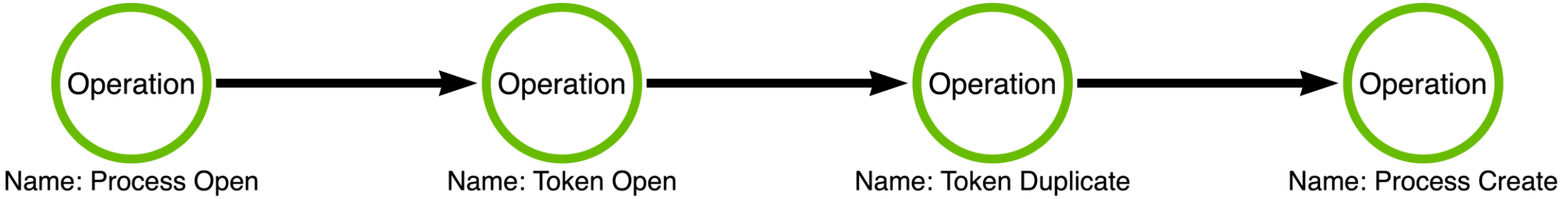
Simplification

- The following operations can be dropped as they are not necessary to make the operation chain valid:
  - Handle Close
    - This is used to close the handle to the duplicate copy of the token.
  - Handle Close
    - This is used to close the handle to the original copy that was generated via OpenProcessToken.
  - Handle Close
    - This is used to close the handle to the process that was generated via OpenProcess.



Morphological Analysis – Vulpes Vulpes

- Fox
  - Family: Canidae
  - Genus: Vulpes
  - Species: Vulpes Vulpes
- The change from Thread Set to Process Create is a significant one.
  - Detections focused on finding threads that are performing impersonation fail.



Sample 6

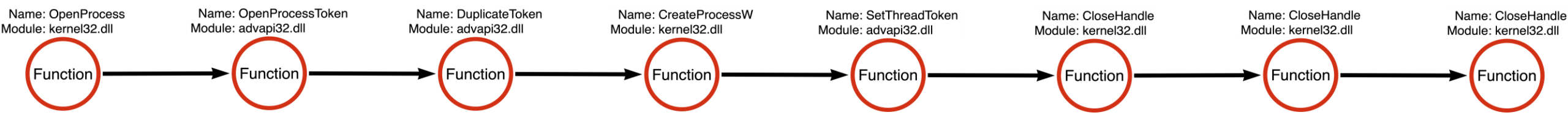
Sample 6: Source Code

```
12      hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, true, PID);
13
14      OpenProcessToken(hProcess, TOKEN_DUPLICATE, &hToken);
15      DuplicateToken(hToken, SecurityImpersonation, &hDuplicate);
16
17      STARTUPINFO si = {};
18      si.cb = sizeof si;
19      PROCESS_INFORMATION pi = {};
20
21      CreateProcessW(L"C:\\Windows\\System32\\cmd.exe", NULL, NULL, NULL, TRUE, CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
22      SetThreadToken(&pi.hThread, hDuplicate);
23
24      CloseHandle(hDuplicate);
25      CloseHandle(hToken);
26      CloseHandle(hProcess);
```


https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%206/src/Source.cpp

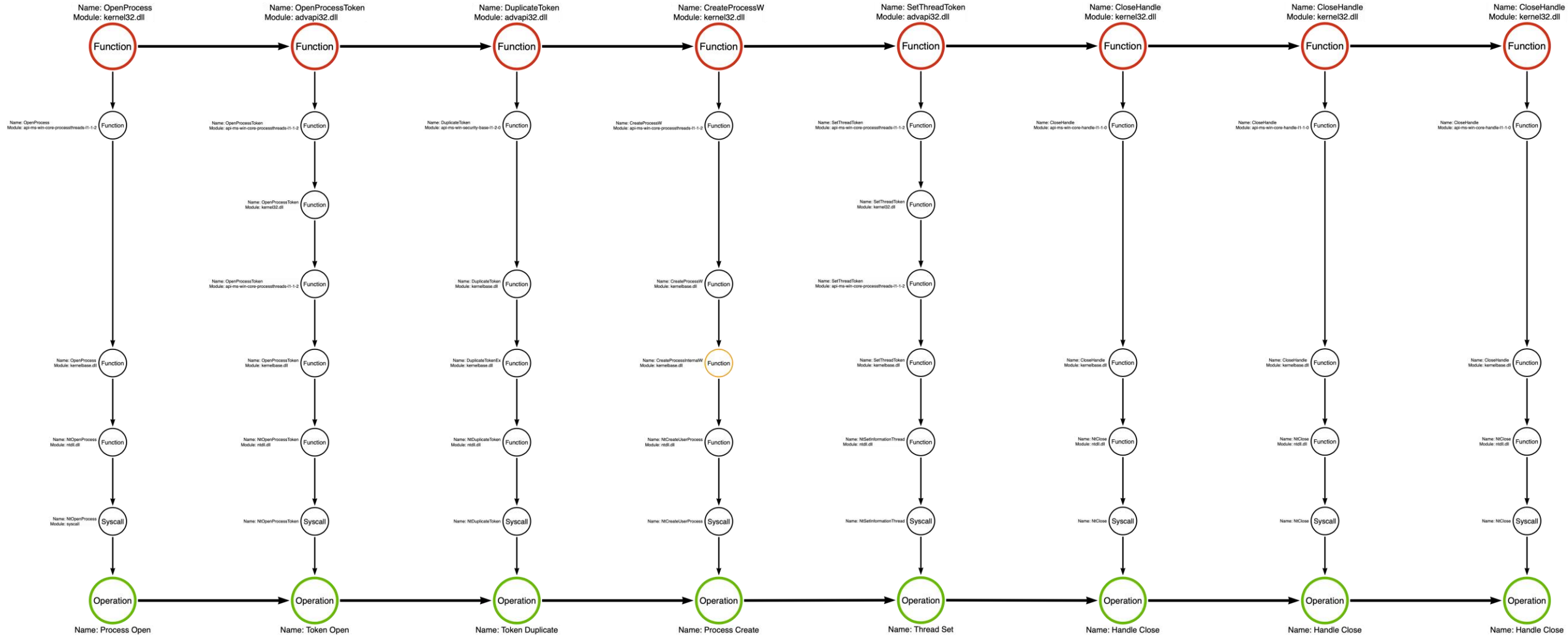
Sample 6: Source Code

```
12      hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, true, PID);
13
14      OpenProcessToken(hProcess, TOKEN_DUPLICATE, &hToken);
15      DuplicateToken(hToken, SecurityImpersonation, &hDuplicate);
16
17      STARTUPINFO si = {};
18      si.cb = sizeof si;
19      PROCESS_INFORMATION pi = {};
20
21      CreateProcessW(L"C:\\Windows\\System32\\cmd.exe", NULL, NULL, NULL, TRUE, CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
22      SetThreadToken(&pi.hThread, hDuplicate);
23
24      CloseHandle(hDuplicate);
25      CloseHandle(hToken);
26      CloseHandle(hProcess);
```



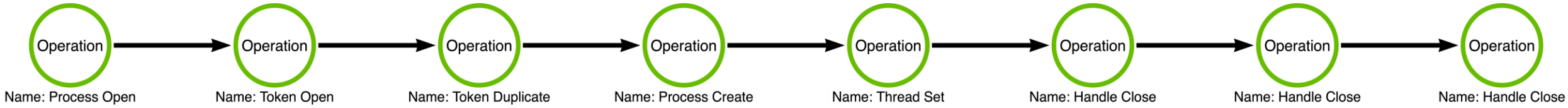
https://github.com/jaredcatkinson/MalwareMorphology/blob/main/Sample%206/src/Source.cpp

Tool Graph: Sample 6



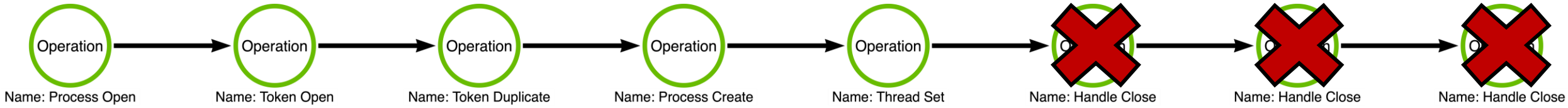
Operation Chain

- Sample 6 implements a novel variation proposed by Jonathan Johnson.
  - It begins in a similar fashion to Sample 5 by finding a token to impersonate.
  - However, when it creates a process, it does so without specifying the token.
    - This eliminates a detection opportunity where processes spawned in a higher user context than their parents are considered suspicious.
  - To complete the chain, the primary thread of the new process is then told to impersonate which achieves the same outcome in a slightly more difficult to detect manner.



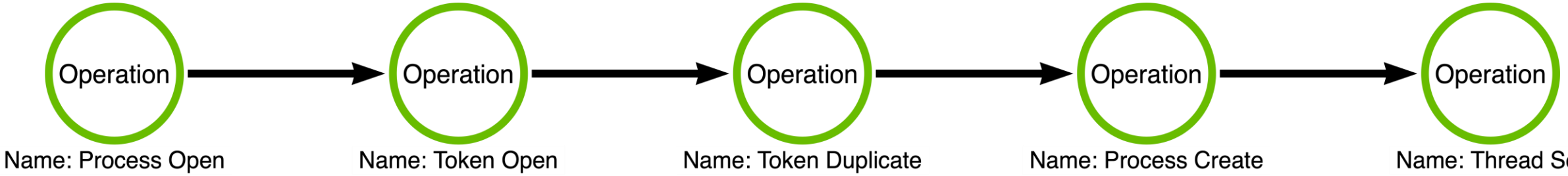
Simplification

- The following operations can be dropped as they are not necessary to make the operation chain valid:
  - Handle Close
    - This is used to close the handle to the duplicate copy of the token.
  - Handle Close
    - This is used to close the handle to the original copy that was generated via OpenProcessToken.
  - Handle Close
    - This is used to close the handle to the process that was generated via OpenProcess.



Morphological Analysis – Chrysocyon Brachyurus

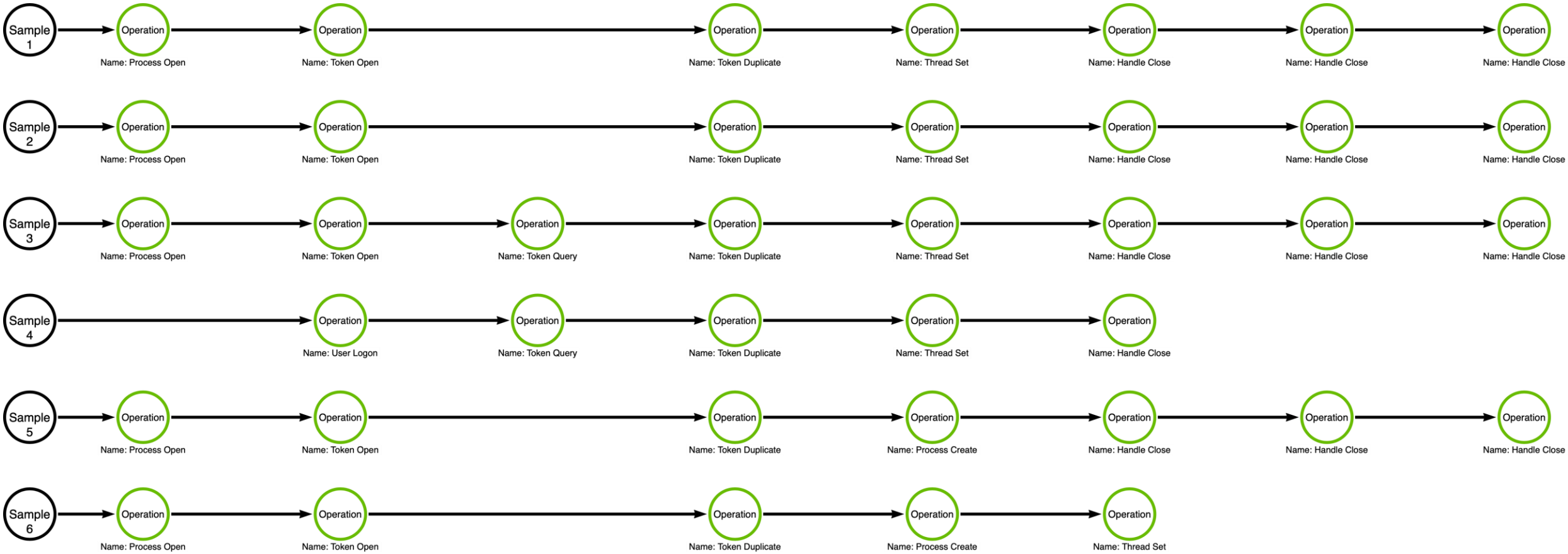
- Maned Wolf
  - Family: Canidae
  - Genus: Chrysocyon
  - Species: Chrysocyon Brachyurus
- A mix between a fox and a wolf.
- Creates a new process in the vain of Sample 5, but then applies the token to a thread in the new process instead of the process itself.



Bringing it All Home

Function Chains

Operation Chains



Morphological View of Simplified Operation Chains


Operation → Name: Process Open
Operation → Name: Token Open
Operation → Name: Token Duplicate
Operation → Name: Thread Set


Operation → Name: Process Open
Operation → Name: Token Open
Operation → Name: Token Duplicate
Operation → Name: Thread Set


Operation → Name: Process Open
Operation → Name: Token Open
Operation → Name: Token Duplicate
Operation → Name: Thread Set

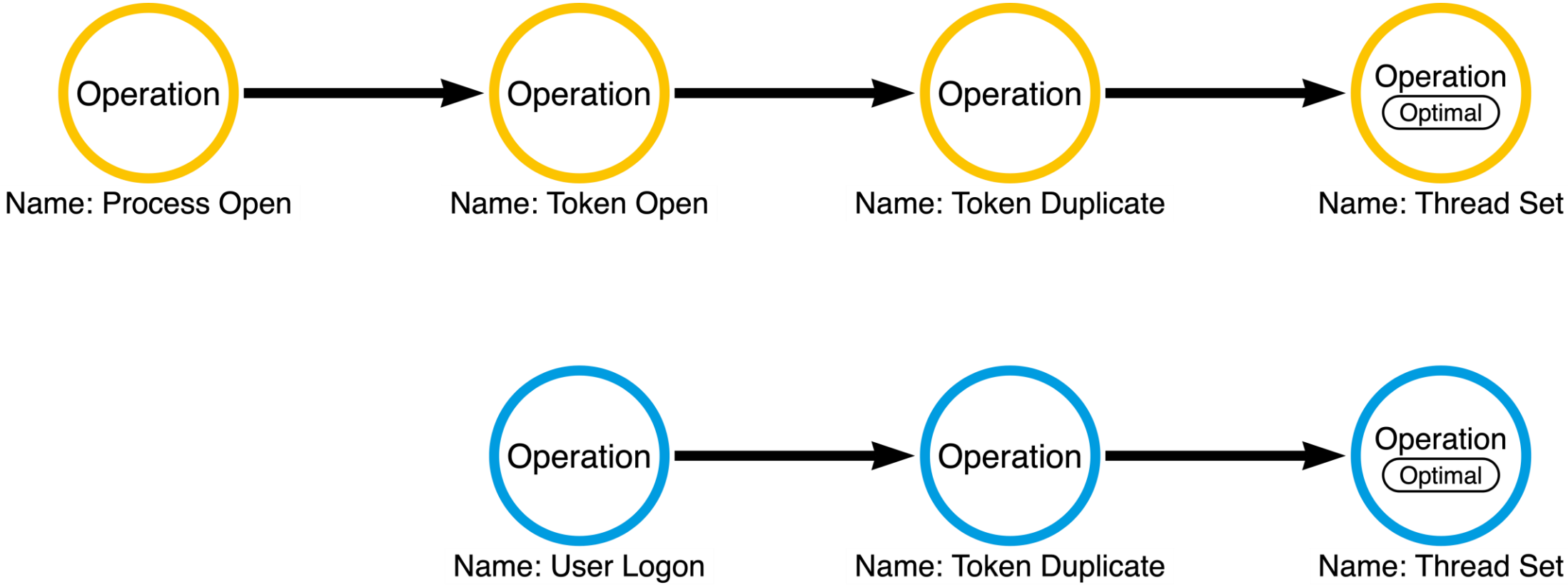

Operation → Name: User Logon
Operation → Name: Token Duplicate
Operation → Name: Thread Set


Operation → Name: Process Open
Operation → Name: Token Open
Operation → Name: Token Duplicate
Operation → Name: Process Create

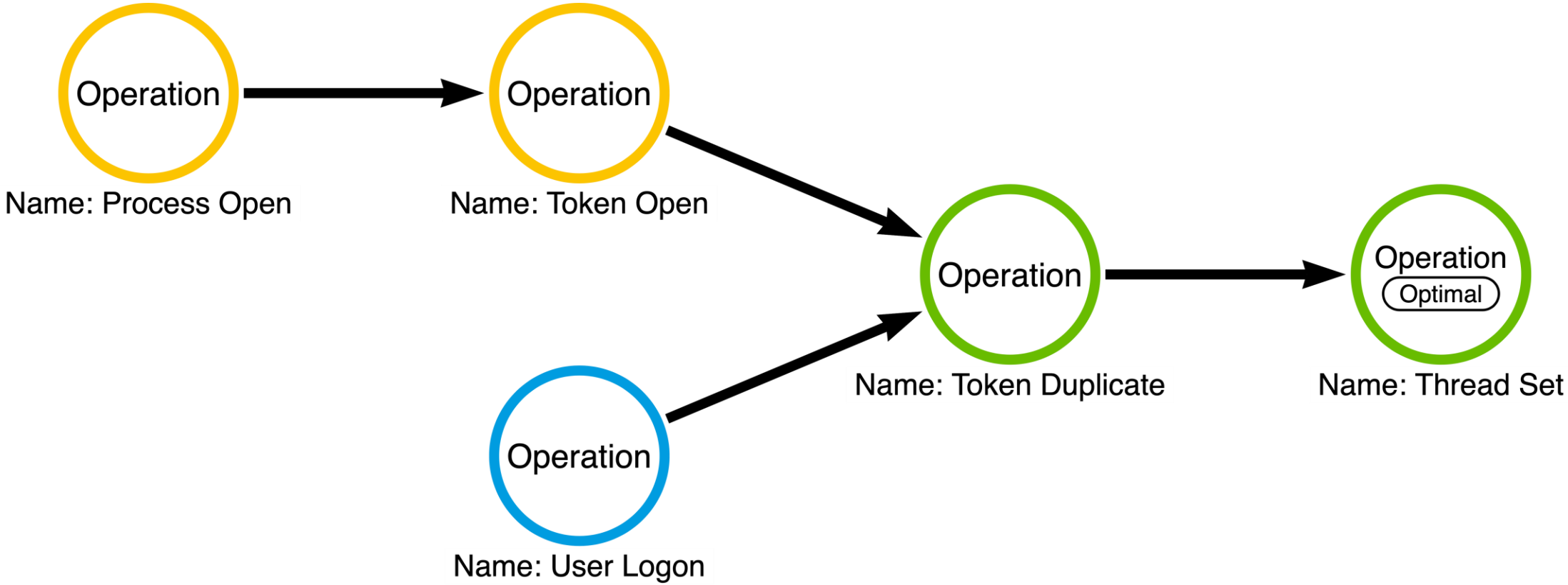

Operation → Name: Process Open
Operation → Name: Token Open
Operation → Name: Token Duplicate
Operation → Name: Process Create
Operation → Name: Thread Set

Intra-Chain Detections

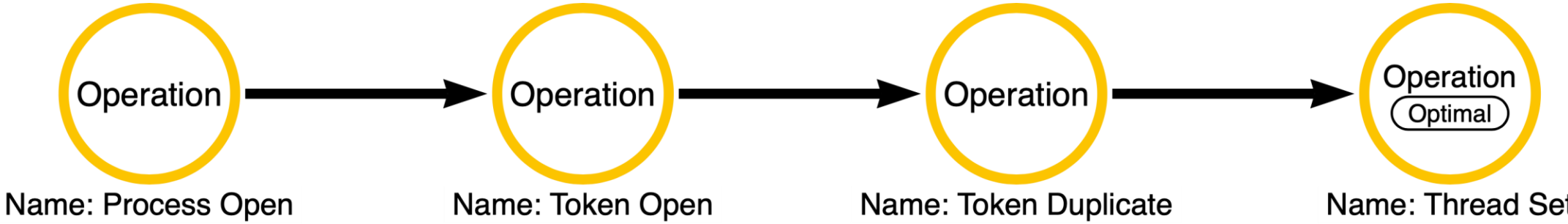
Example 1: Late Chain Overlap

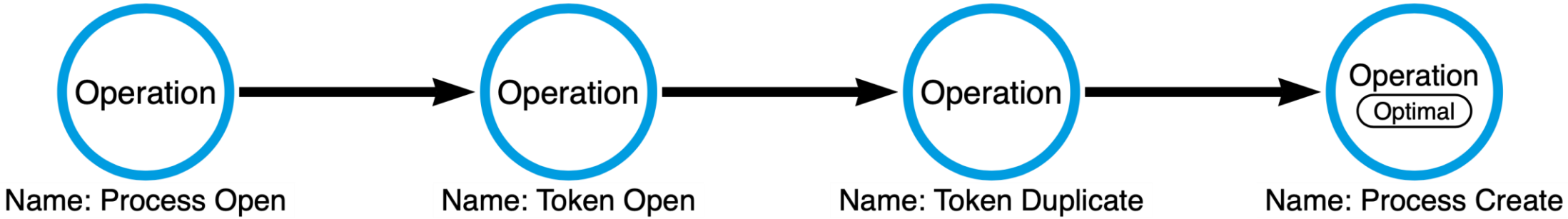


Example 1: Late Chain Overlap

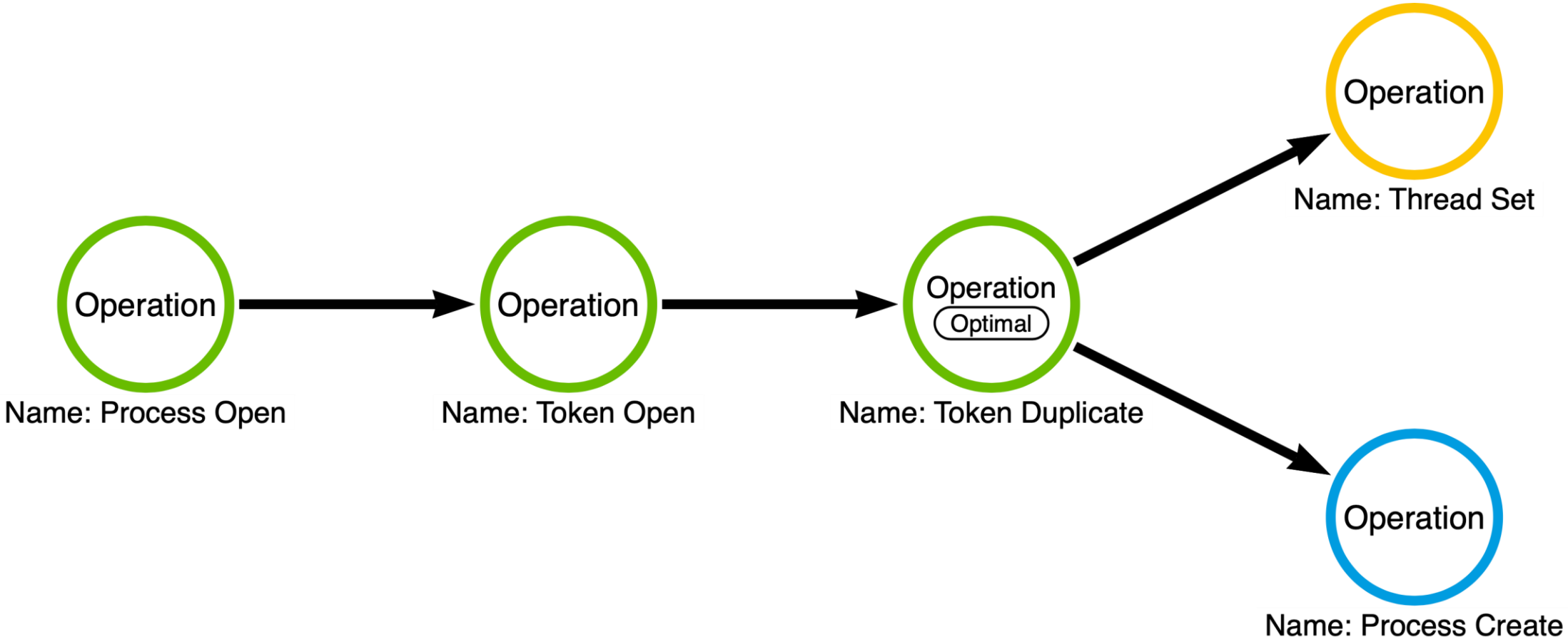


Example 2: Early Chain Overlap

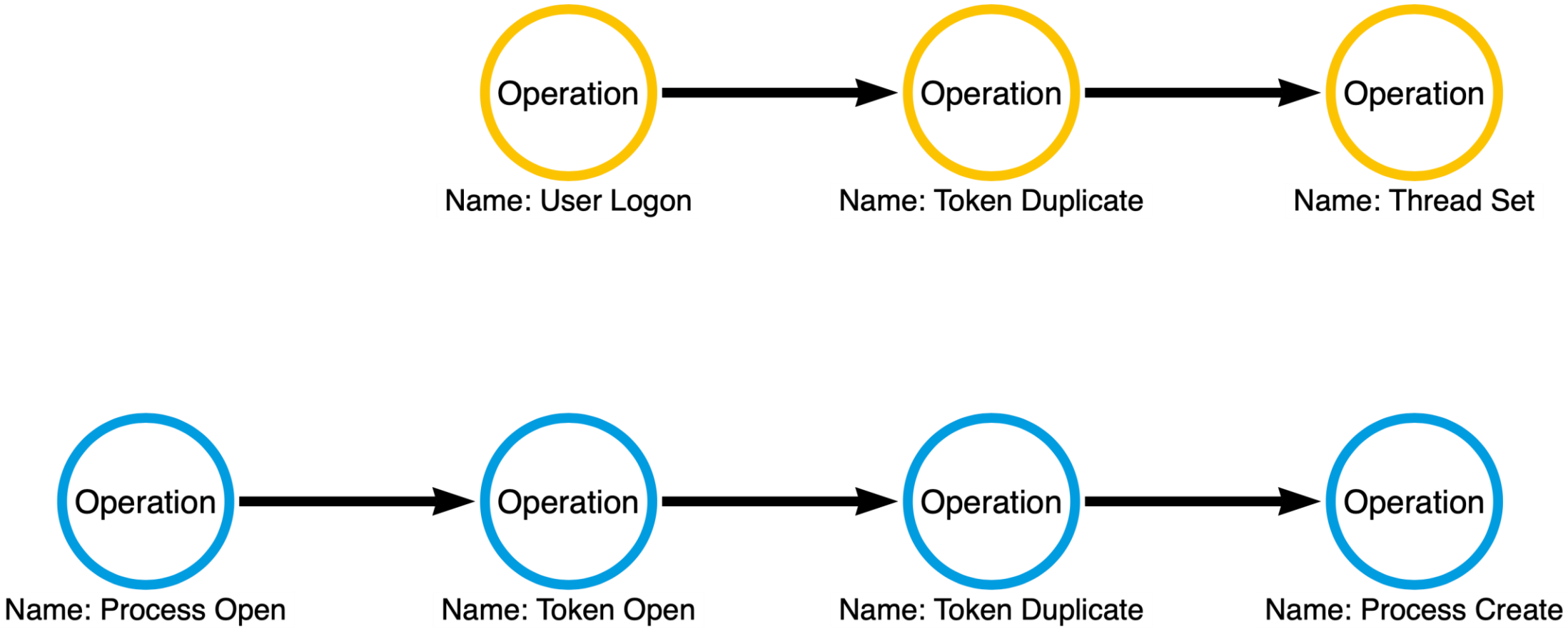




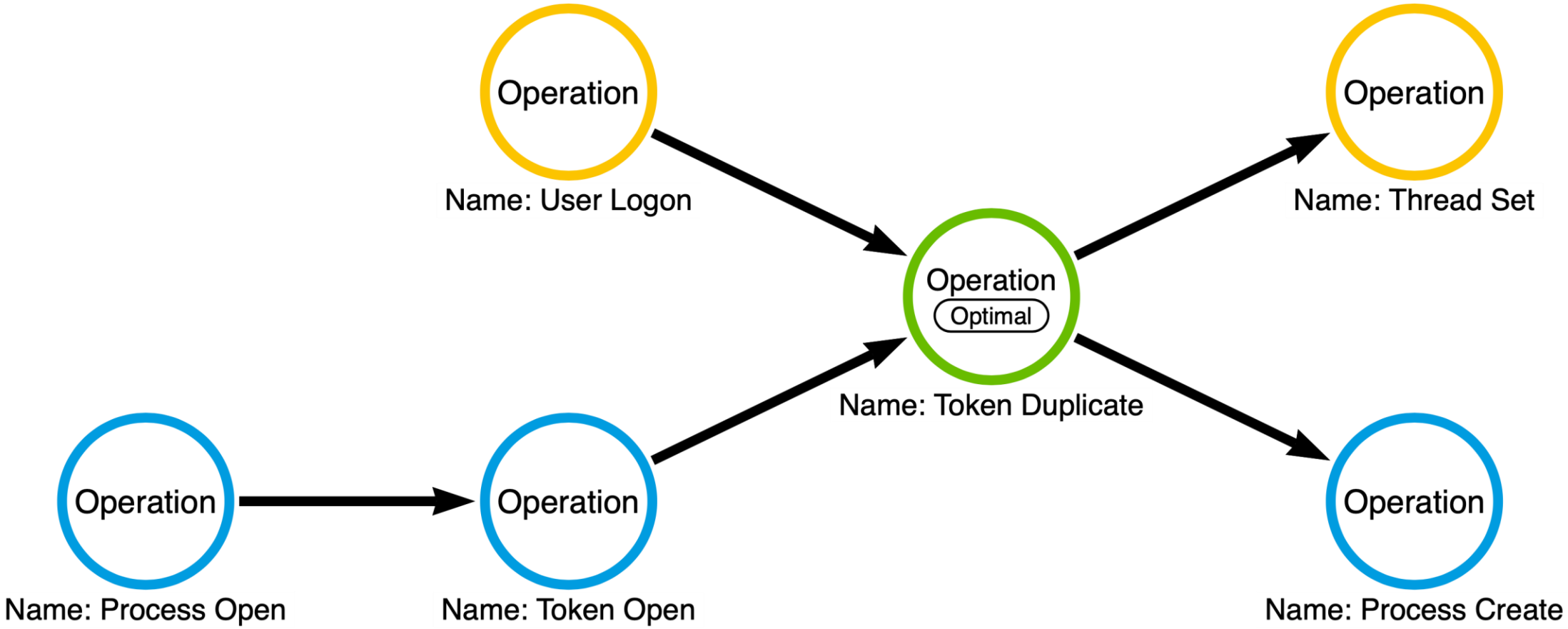
Example 2: Early Chain Overlap



Example 3: Mid Chain Overlap



Example 3: Mid Chain Overlap



The Benefits of Operational Focused Detection

- Brings confidence in the detection
- Applies understanding of the OS the detection is applied to
- Highlights detections are dependent on variational change.
- Detections are objectively viewed (server side/mix of client & server side)
- Identification of events that are directly related to activity

Questions?

Thank you for joining me today! We hope that you enjoyed the workshop and that the contents will help you make more intentional decisions whether you are a Detection Engineer, CTI Analyst, Red Teamer, Purple Teamer, or Security Architet!