x86 실행파일에서 악성코드의 행동을 정적으로 찾아내기

이승중

2012.01.17

서울대학교 프로그래밍 연구실

악성코드의 진화

- 새로운 악성코드의 끊임없는 출현

- 진단/분석 을 피하기 위한 여러 방법 도입
  - 코드 난독화
  - 디버깅 무력화
  - 가상 PC 실행 탐지

코드 난독화의 예

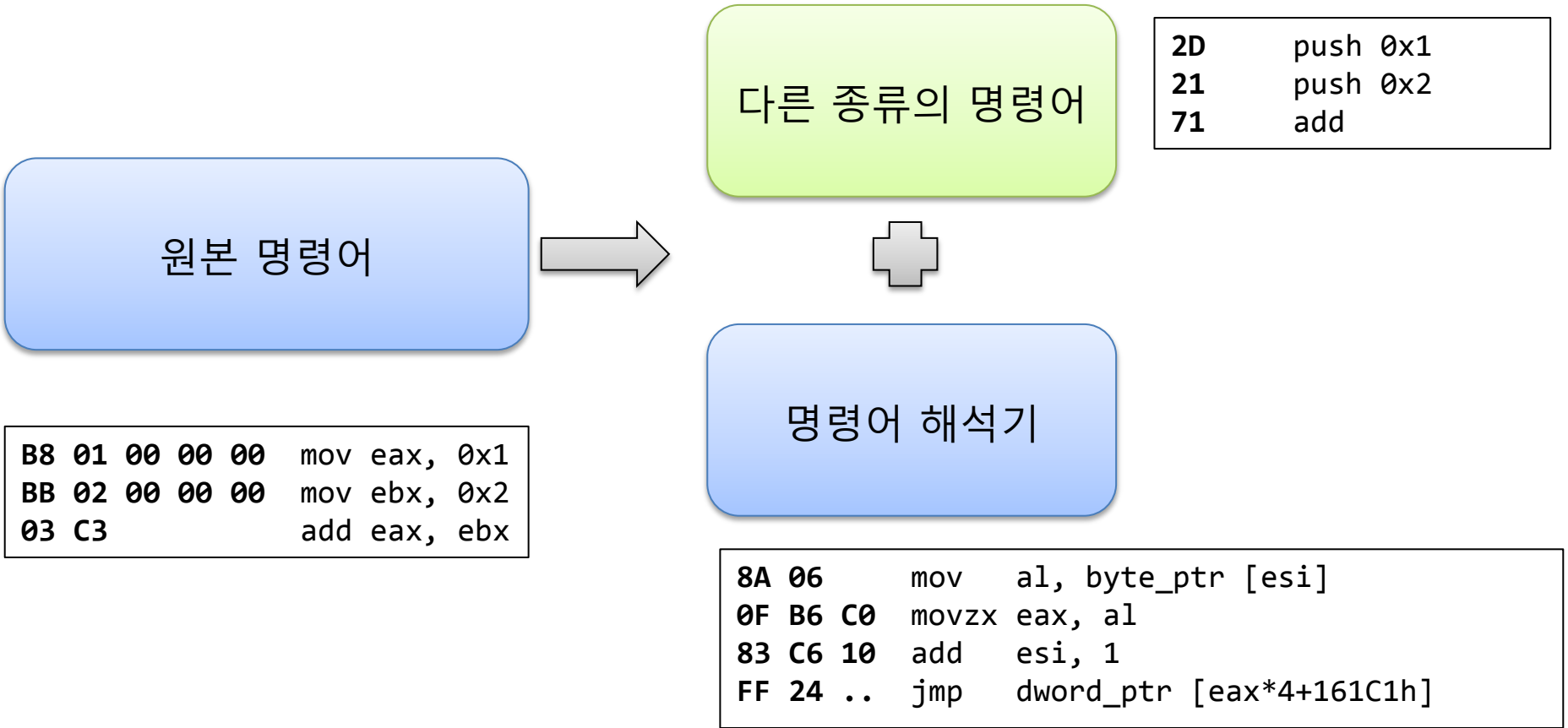
- 가상 머신 난독화 (VM-obfuscation)

원본 명령어

```
B8 01 00 00 00  mov eax, 0x1
BB 02 00 00 00  mov ebx, 0x2
03 C3           add eax, ebx
```

코드 난독화의 예

- 가상 머신 난독화 (VM-obfuscation)



코드 난독화의 예

- 가상 머신 난독화 (VM-obfuscation)
  - 역공학을 불가능하게 만드는 것이 목적
    - 일반 프로그램의 시리얼 입력 부분 등에 쓰임
  - VMProtect, Themida

악성코드 검출 방식의 한계

- 프로그램의 모양을 살펴보는 방식
  - 어느 부분을 살펴야 할지 모르므로
    악성코드에 대한 분석 선행
  - 각종 난독화 방법에 약함

virustotal

SHA256: 293952a4cad58d43a621258255dcf22f5cd71ec4d70a6c915e8cc0d7dd37ac01

Detection ratio: 24 / 43

Analysis date: 2011-11-06 06:25:29 UTC (2 months, 1 week ago)

Antivirus	Result	Update
AhnLab-V3	-	20111105
AntiVir	W32/Dahorse.3.C	20111104
Antiy-AVL	-	20111106
Avast	-	20111105
AVG	Win32/Dahorse.A	20111105
BitDefender	Win32.Dahorse.A	20111106
ByteHero	-	20111104
CAT-QuickHeal	-	20111105
ClamAV	-	20111106
Commtouch	W32/TrojanX.JYJ	20111105
Comodo	UnclassifiedMalware	20111106
DrWeb	-	20111106
Emsisoft	Virus.Win32.Dahorse!IK	20111106
eSafe	Win32.TrojanHorse	20111102
eTrust-Vet	-	20111105
F-Prot	W32/TrojanX.JYJ	20111105
F-Secure	Win32.Dahorse.A	20111106
Fortinet	W32/Dahorse.A	20111106

virustotal

SHA256: d712edfc38a304e48647bf63930a62e3c41a39953c876d08ab53c7908a5178b9

Detection ratio: 0 / 43

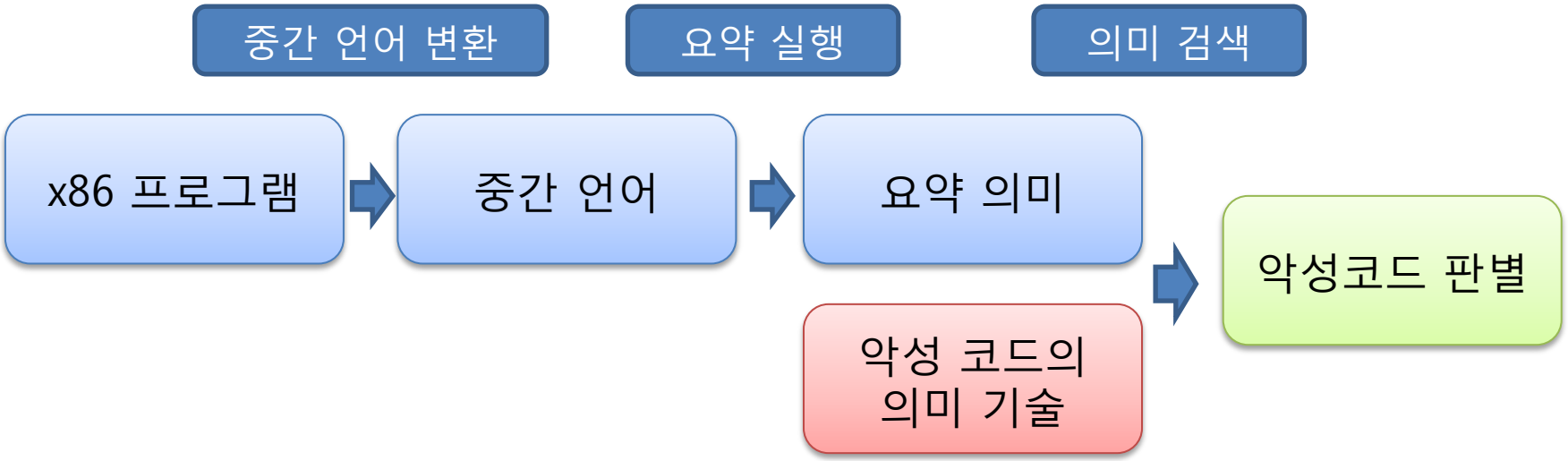
Analysis date: 2012-01-16 17:42:34 UTC (1 minute ago)

Antivirus	Result	Update
AhnLab-V3	-	20120116
AntiVir	-	20120116
Antiy-AVL	-	20120116
Avast	-	20120116
AVG	-	20120116
BitDefender	-	20120116
ByteHero	-	20120111
CAT-QuickHeal	-	20120116
ClamAV	-	20120116
Commtouch	-	20120116
Comodo	-	20120116
DrWeb	-	20120116
Emsisoft	-	20120116
eSafe	-	20120115
eTrust-Vet	-	20120116
F-Prot	-	20120116
F-Secure	-	20120116
Fortinet	-	20120115

악성코드를 검출하는 다른 방법

- 프로그램의 실행의미를 본다
  - x86 프로그램의 실행의미는 무엇인지
  - 악성코드들은 어떤 실행의미를 가지고 있는지

실행의미 기반 악성 코드 검출



x86 중간언어

```
add eax, 0x1
```

적은 수의 같은 기능을 하는 명령어로 표현

```
t2 := EAX;
t0 := (t2 + 1);
CC_OP := 3;
CC_DEP1 := t2;
CC_DEP2 := 1;
CC_NDEP := 0;
T_0 := (t2 + 1);
CF := (T_0 < t2);
T_1 := (T_0 & 255);
PF := (~(((((T_1 >> 7) ^ (T_1 >> 6)) ^ ((T_1 >> 5) ^ (T_1 >> 4))) ^ (((T_1 >> 3) ^ (T_1 >> 2)) …
AF := (1 == (16 & (T_0 ^ (t2 ^ 1)))));
ZF := (T_0 == 0);
SF := (1 == (1 & (T_0 >> 31)));
OF := (1 == (1 & (((t2 ^ (1 ^ -1)) & (t2 ^ T_0)) >> 31)));
EFLAGS := ((EFLAGS & (-2 & -5)) & ((-17 & (-65 & -129)) & -2049));
EFLAGS := ((EFLAGS | ((CF << 0) | (PF << 2))) | (((AF << 4) | ((ZF << 6) | (SF << 7))) …
EAX := t0;
```

x86 중간언어

$$
\begin{array}{lcll}
C & ::= & \texttt{mov}_n\ E\ E & \\
 & | & \texttt{jmpnz}\ E\ E & // \text{ conditional jump} \\
 & | & \texttt{save}_n\ E\ E & // \text{ memory operation} \\
E & ::= & x & // \text{ variable (register, flag)} \\
 & | & \texttt{new} & // \text{ allocate new memory section} \\
 & | & n & // \text{ integer, address} \\
 & | & [E]_n & // \text{ load memory} \\
 & | & \star\ E & // \text{ unary operator} \\
 & | & E\ \diamond\ E & // \text{ binary operator}
\end{array}
$$

x86 프로그램의 의미구조

$$
\begin{aligned}
(\sigma,\ m)\ \text{as}\ st \in \quad & State &&= && Env \times Mem \\
\sigma \in \quad & Env &&= && Var \xrightarrow{\text{fin}} Value \\
m \in \quad & Mem &&= && Addr \xrightarrow{\text{fin}} Value \\
v \in \quad & Value &&= && \mathbb{Z} + Addr
\end{aligned}
$$

$$[\![\cdot]\!] : Program \to Trace$$

$$[\![(a_1,\ b_1)(a_2,\ b_2)..., a]\!] \ = \ st_0 st_1 ... st_m$$

where

$$st_0 = \langle\ 0,\ \{\}, \{a_1 \mapsto (b_1,\ 0),\ a_2 \mapsto (b_2,\ 0), ...\ \},\ a \rangle$$

$$st_i \to st_{i+1}$$

프로그램 실행 결과는 일련의 상태전이

x86 프로그램의 의미구조

$$[\![\cdot]\!] : Program \rightarrow Trace$$

$$[\![(a_1,\ b_1)(a_2,\ b_2)..., a]\!] \ = \ st_0 st_1 ... st_m$$

where

$$st_0 = \langle\ 0,\ \{\}, \{a_1 \mapsto (b_1,\ 0),\ a_2 \mapsto (b_2,\ 0), ...\ \},\ a \rangle$$

$$st_i \rightarrow st_{i+1}$$

X86 프로그램의 요약 의미구조

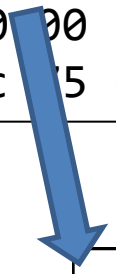
- 요약방법

  프로그램 카운터가 실행 중 가질 수 있는 모든 메모리 주소에 대해

  가질 수 있는 모든 메모리 상태, 레지스터의 값들을 요약해서 저장함

X86 프로그램

```
0x013513BE • b8 01 00 00 00•bb 02 00 00 00•83 c0 01•8a 06•0f
0x013513CE   b6 c0•83 c6 01 ff 24 85 c1 61 01 00 5f 5e 5b 81
0x013513DE   c4 c0 00 00 00 3b ec e8 47 fd ff ff 8b e5 5d c3
0x013513EE   cc cc 75 01 c3 55 8b ec 83 ec 00 50 52 53 56 57
```



환경

레지스터	값
esp	{0x0026fe88}
eax	{0x005513b8}
…	…

메모리

주소	값
0x0026fe88	…
0x0026fe8c	{0x11223344}
…	…

요약 실행기

$$[\![\cdot]\!] : Program \rightarrow \hat{Trace}$$

$$[\![(a_1,\ b_1)(a_2,\ b_2)...,a]\!] = fix(\hat{F} \stackrel{\text{def}}{=} \lambda \hat{T}.\{\hat{st}_0\} \cup \hat{Next}\ \hat{T})$$

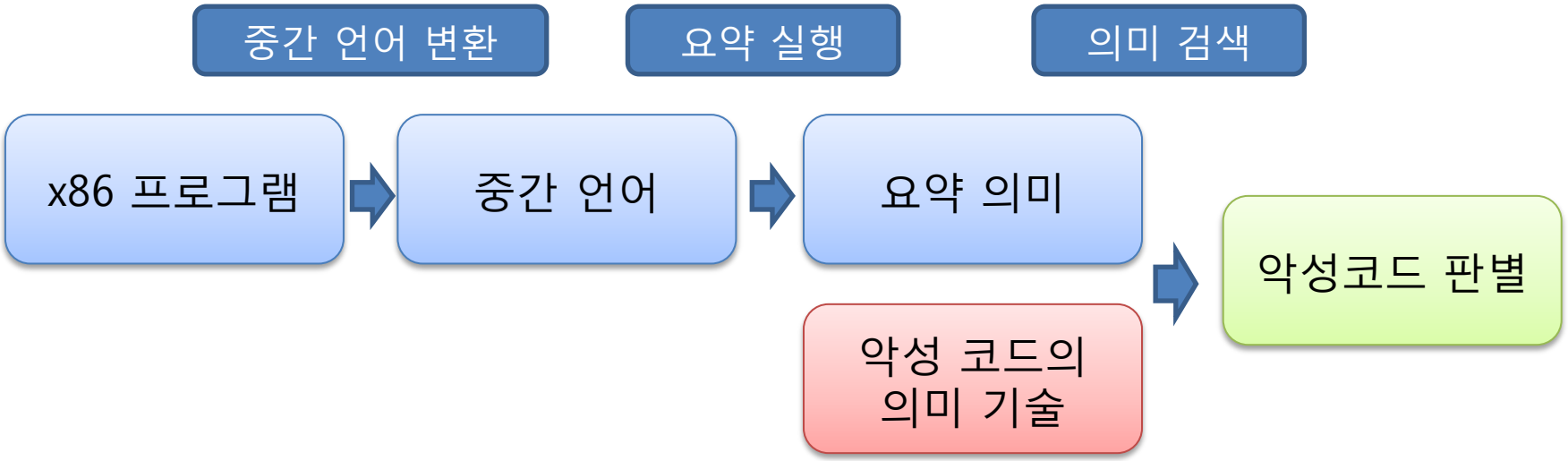
where

$$\hat{st}_0 = \langle\ \{\}, \{a_1 \mapsto \{(b_1,\ 0)\},\ a_2 \mapsto \{(b_2,\ 0)\}, ...\ \},\ \{a\}\rangle$$

$$\begin{aligned} \hat{Next} &= (\rho\ \sqcup) \circ \hat{\pi} \circ (\rho_{\cup}\ \hat{next}) \\ \hat{next} &= \lambda \hat{st}.\{\hat{st'} | \hat{st} \hat{\rightarrow} \hat{st'}\} \end{aligned}$$

- 입력은 x86 프로그램
- 결과는 (프로그램 카운터, 요약된 상태) 들의 집합

실행의미 기반 악성 코드 검출

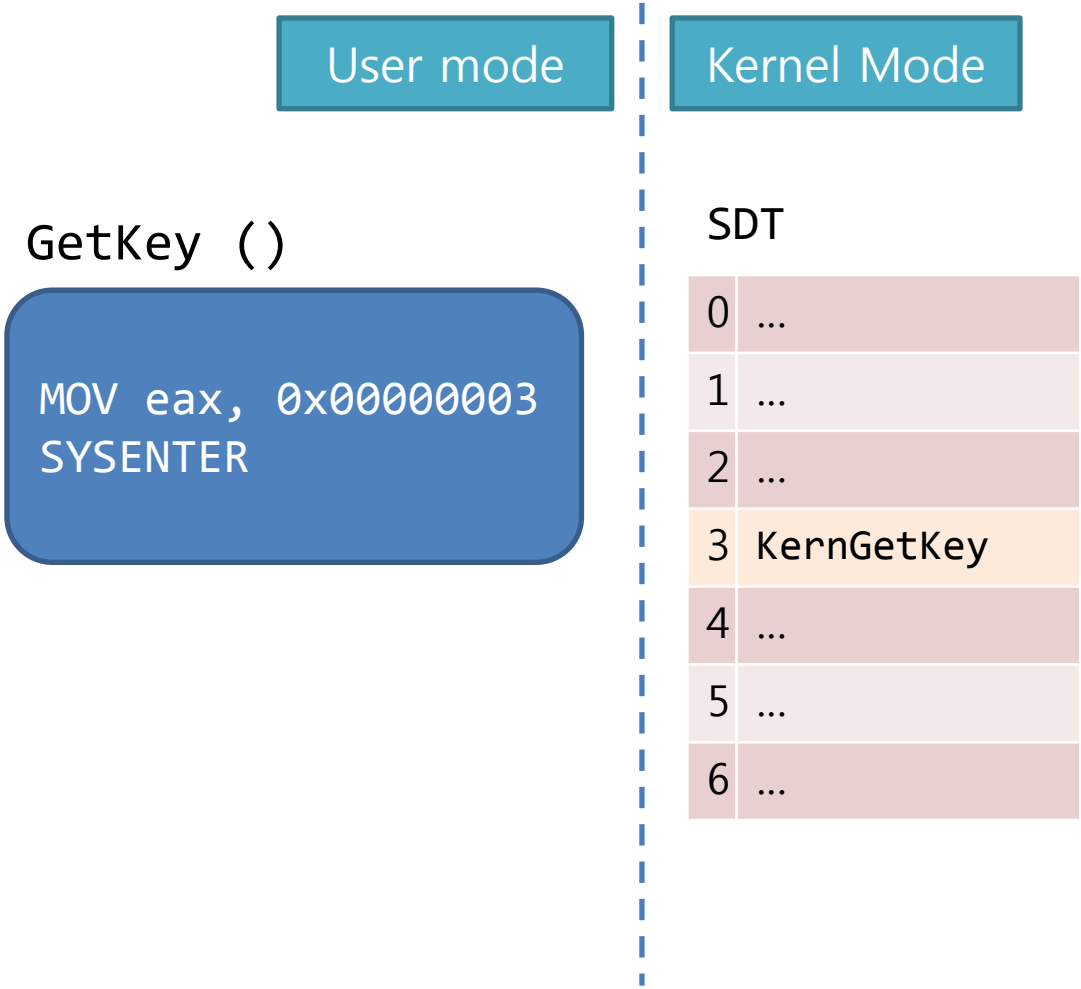


악성코드의 의미기술

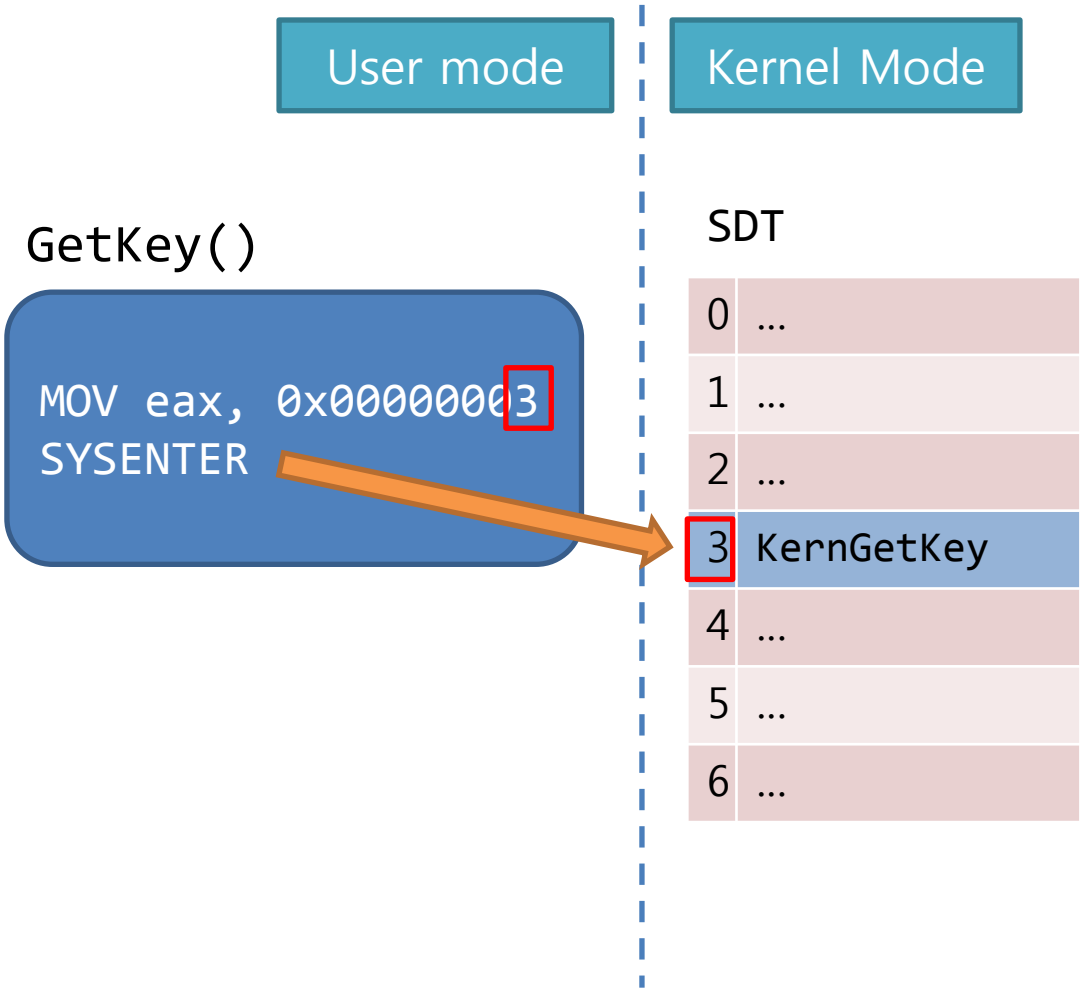
- 어떤 것을 악성코드의 행동이라고 할까
  - 시스템 행동 가로채기

- 많은 악성코드들이 시스템 행동을 가로챈다
  - 키보드 로거: 키보드 인터럽트 행동 가로채기
  - Botnet: 네트워크, 파일시스템 가로채기

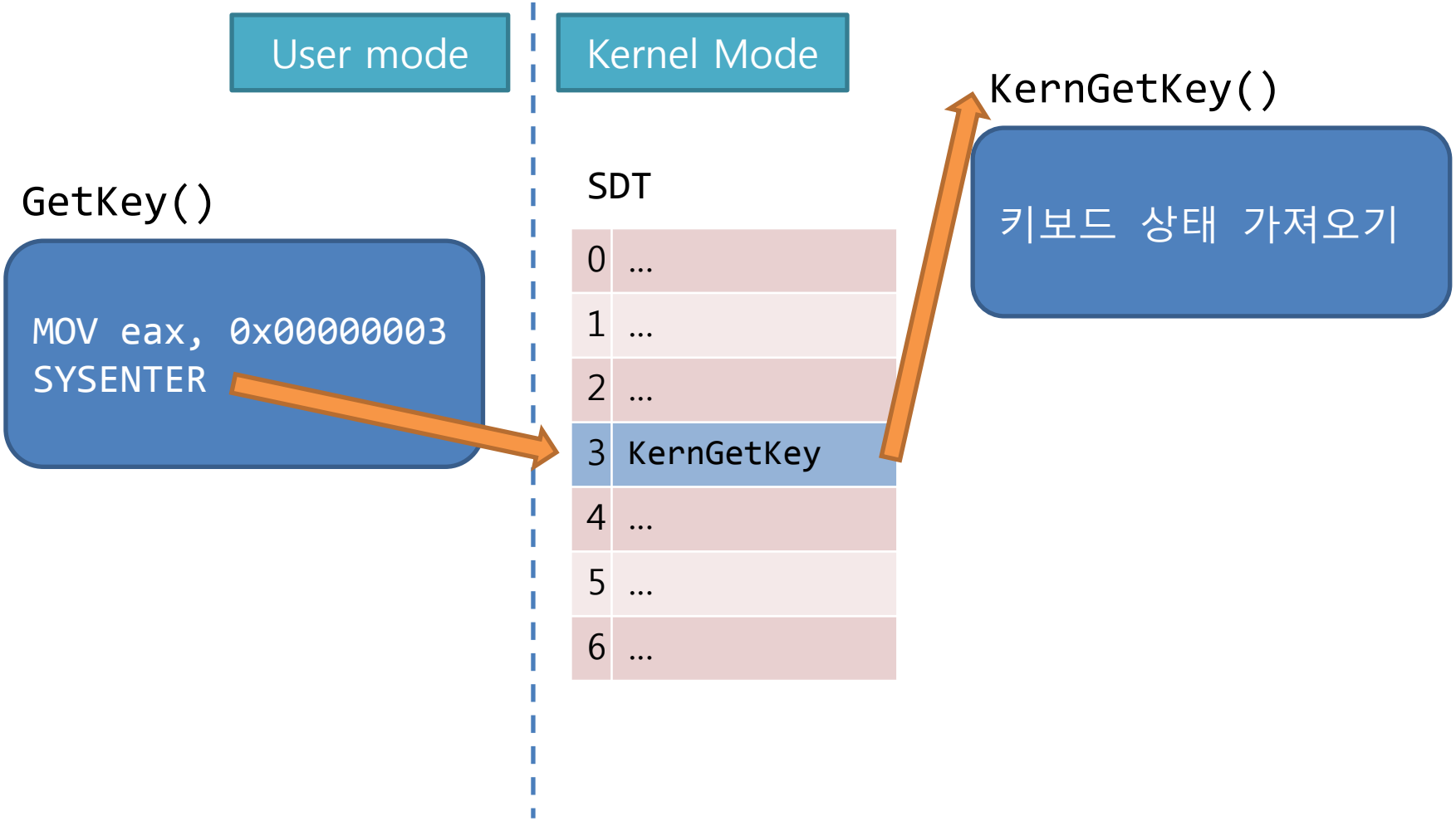
SDT 가로채기


User mode
Kernel Mode
GetKey ()
MOV eax, 0x00000003
SYSENTER
SDT
0 …
1 …
2 …
3 KernGetKey
4 …
5 …
6 …

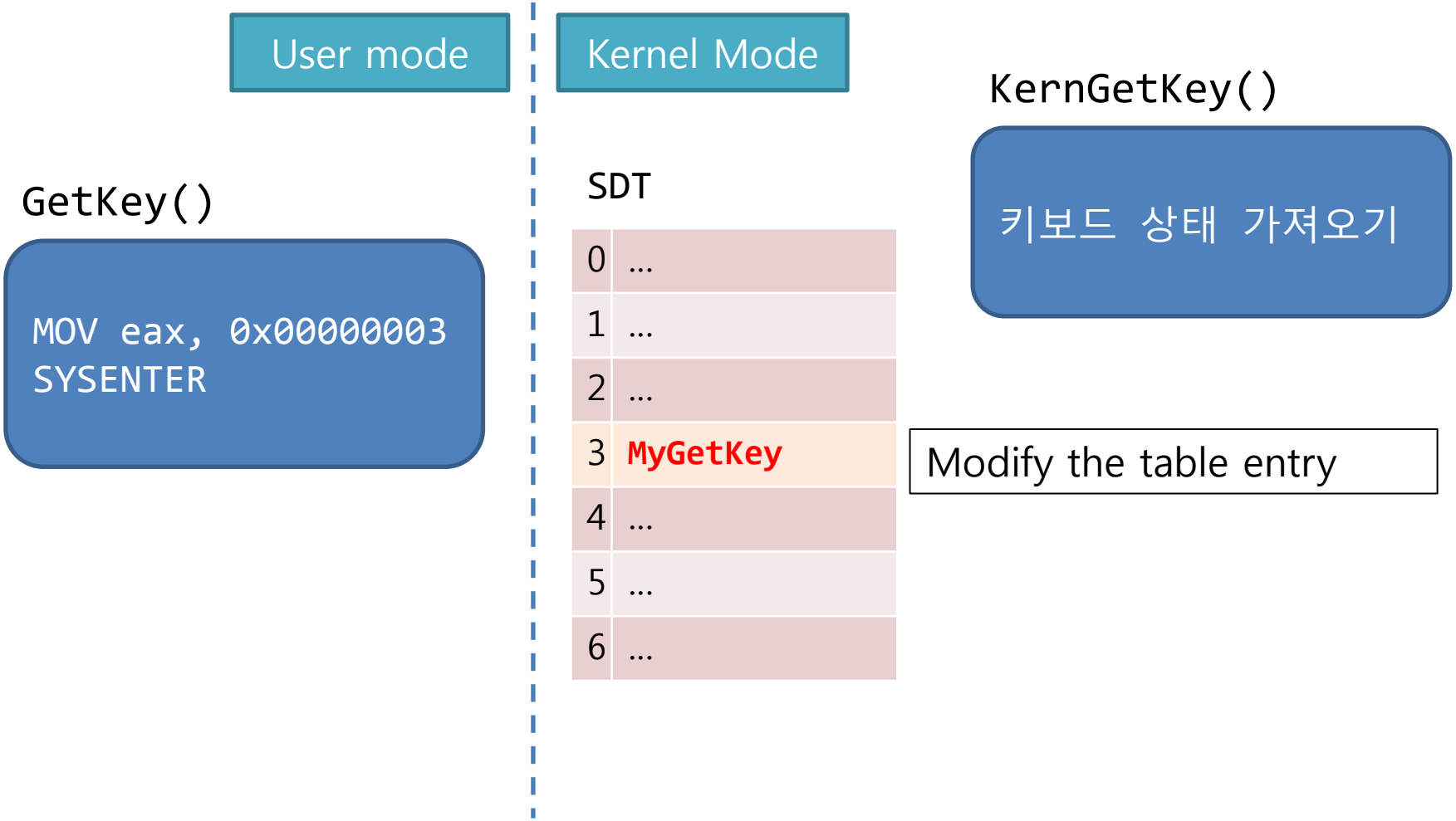
SDT 가로채기


User mode
Kernel Mode
GetKey()
MOV eax, 0x00000003
SYSENTER
SDT
0 …
1 …
2 …
3 KernGetKey
4 …
5 …
6 …

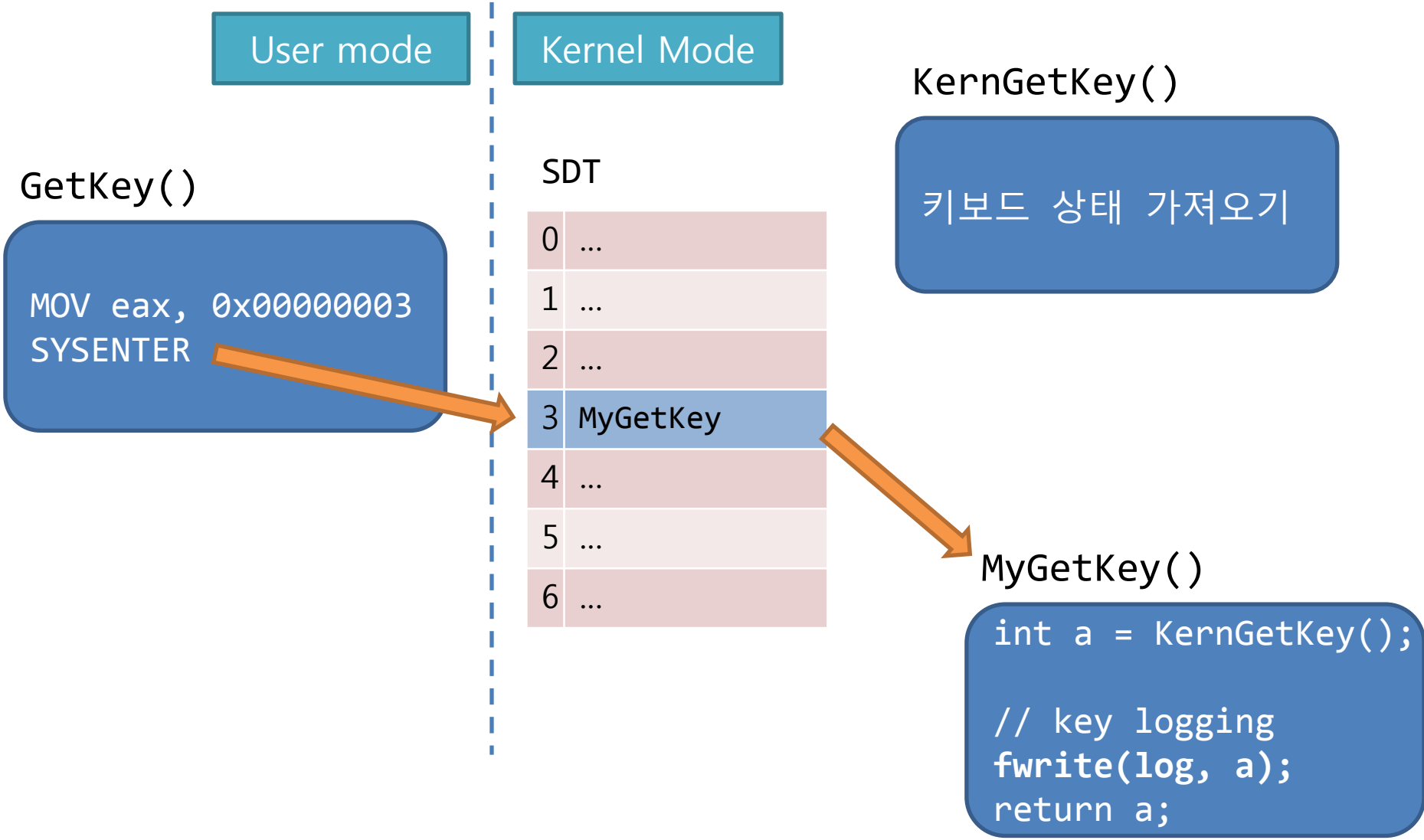
SDT 가로채기


User mode
Kernel Mode
KernGetKey()
GetKey()
SDT
키보드 상태 가져오기
MOV eax, 0x00000003
SYSENTER
0 …
1 …
2 …
3 KernGetKey
4 …
5 …
6 …

SDT 가로채기


User mode
Kernel Mode
KernGetKey()
키보드 상태 가져오기
GetKey()
MOV eax, 0x00000003
SYSENTER
SDT
0 …
1 …
2 …
3 MyGetKey
4 …
5 …
6 …
Modify the table entry

SDT 가로채기


User mode
Kernel Mode
KernGetKey()
키보드 상태 가져오기
GetKey()
MOV eax, 0x00000003
SYSENTER
SDT
0 …
1 …
2 …
3 MyGetKey
4 …
5 …
6 …
MyGetKey()
int a = KernGetKey();
// key logging
fwrite(log, a);
return a;

다양한 가로채기 방법

악성코드	가로채기 방법
Troj/Keylogg-LF	SetWindowsHookEx
Troj/Thief	SetWindowsHookEx
AFXRootkit	WriteProcessMemory
CFSD	FltRegisterFilter
Sony Rootkit	KeServiceDescriptorTable
Vanquish	GetProcAddress
Hacker Defender	NtWriteVirtualMemory
Uay backdoor	NdisRegisterProtocol

from hook finder's experimental result
http://bitblaze.cs.berkeley.edu/papers/hookfinder_ndss08.pdf

악성코드의 행동을 표현

c	$:=$	b	
	$\mid$	$b \wedge b$	
b	$:=$	$InsideCodeSection(e)$	address belongs to code section
	$\mid$	$\{e\ (e^*)\}$	function call is in the code
	$\mid$	$\{e := e\}$	assignment is in the code
	$\mid$	$\{x86\_instruction\}$	x86 instruction is in the code
	$\mid$	$\{x86\_instruction\}$ with c	x86 instruction is in the code with given condition
	$\mid$	$e = e$	values of two expressions are equal
	$\mid$	$\neg b$	
	$\mid$	$b \vee b$	
e	$:=$	x	variable
	$\mid$	`r`	register(predefined variable)
	$\mid$	`f`	external function, variable
	$\mid$	_	don't care
	$\mid$	n	integer constant
	$\mid$	s	string
	$\mid$	$e\ (e^*)$	function call
	$\mid$	$e[e]$	array
	$\mid$	$\circ e \mid e \star e$	unary/binary operator

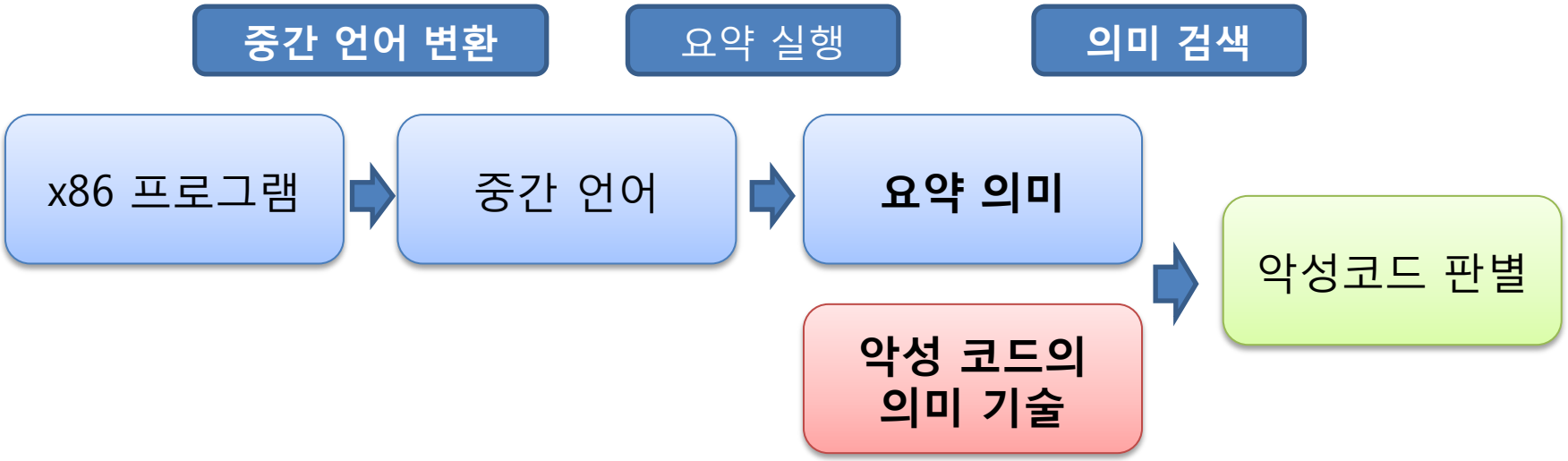
SDT 가로채기 코드

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING theRegistryPath)
{
   …
   // save old system call locations
   OldZwQuerySystemInformation =(ZWQUERYSYSTEMINFORMATION)(SYSTEMSERVICE(ZwQuerySystemInformation));

   // Map the memory into our domain so we can change the permissions on the MDL
   g_pmdlSystemCall = MmCreateMdl(NULL,
                                  KeServiceDescriptorTable.ServiceTableBase,
                                  KeServiceDescriptorTable.NumberOfServices*4);
   if(!g_pmdlSystemCall)
      return STATUS_UNSUCCESSFUL;

   MmBuildMdlForNonPagedPool(g_pmdlSystemCall);

   // Change the flags of the MDL
   g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags | MDL_MAPPED_TO_SYSTEM_VA;

   MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall, KernelMode);

   // hook system calls
   HOOK_SYSCALL( ZwQuerySystemInformation, NewZwQuerySystemInformation, OldZwQuerySystemInformation );

   return STATUS_SUCCESS;
}
```

SDT 가로채기의 표현

$$
\begin{aligned}
& x = \texttt{KeServiceDescriptorTable} \\
\wedge \quad & y = \texttt{MmCreateMdl}(\_,\ x,\ \_,\ \_) \\
\wedge \quad & z = \texttt{MmMapLockedPages}(\_,\ y) \\
\wedge \quad & z[\_] = f \\
\wedge \quad & \mathit{InsideCodeSection}(f)
\end{aligned}
$$

실행의미 기반 악성 코드 검출



목표

- 가상 머신 난독화된 악성코드까지
  - 보다 정교한 분석 필요
  - 상태들을 모을 포인트 정교한 위치 필요
  - 함수, 타입정보들을 유추할 수 있으면 도움

감사합니다