KERNELBLEED

Automatyczne wykrywanie błędów ujawnienia pamięci jądra w systemach Windows i Linux

Mateusz “j00ru” Jurczyk

Security PWNing Conference 2017, Warszawa

Bio

- Security Researcher @ Project Zero, Google
- Wicekapitan @ Dragon Sector
- http://j00ru.vexillium.org/
- @j00ru

#cloudbleed

RTPBleed

Yahoobleed

Optionsbleed

Ticketbleed

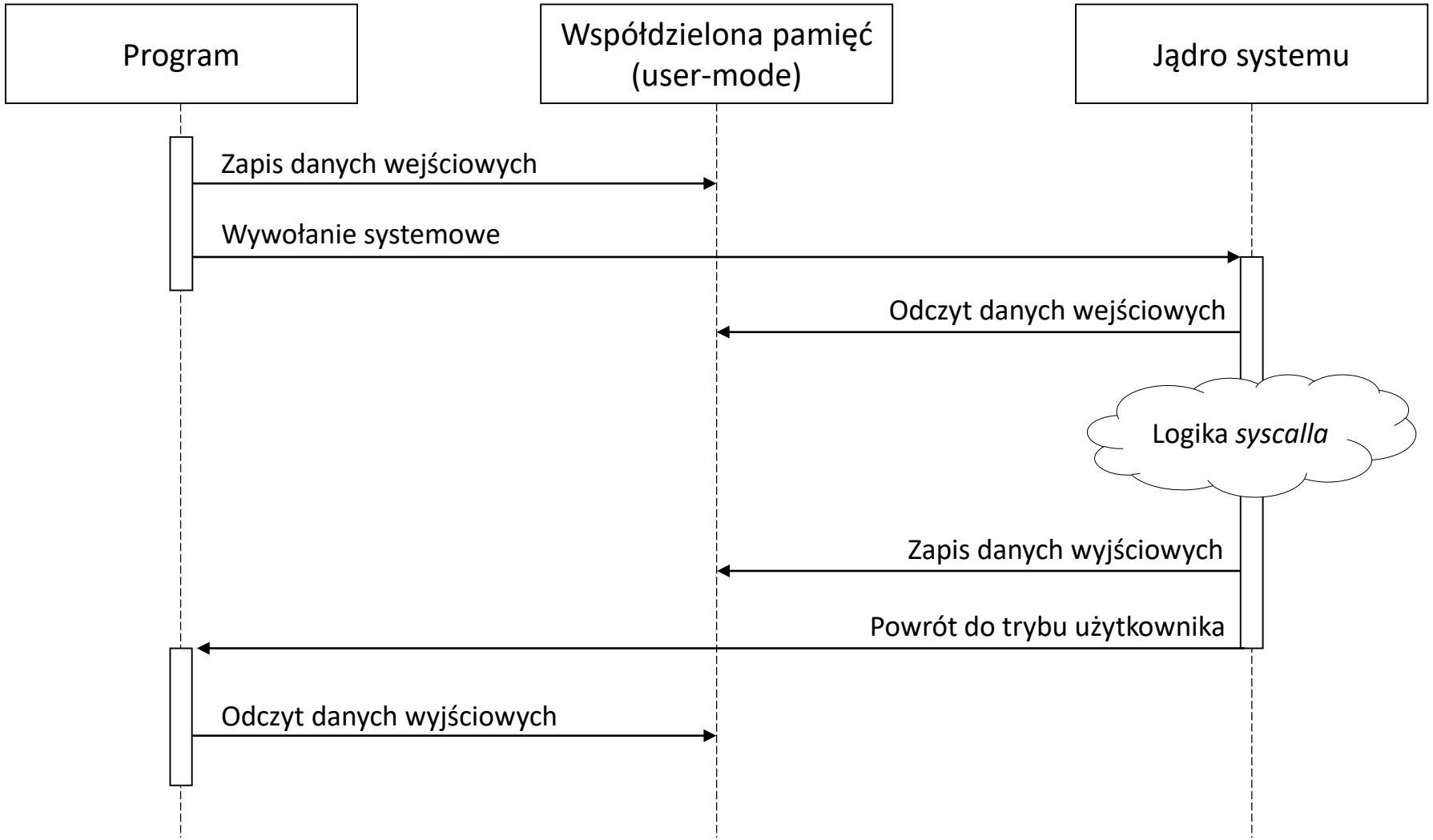
Skąd wycieki danych w jądrach systemów operacyjnych?

Komunikacja aplikacji z jądrem

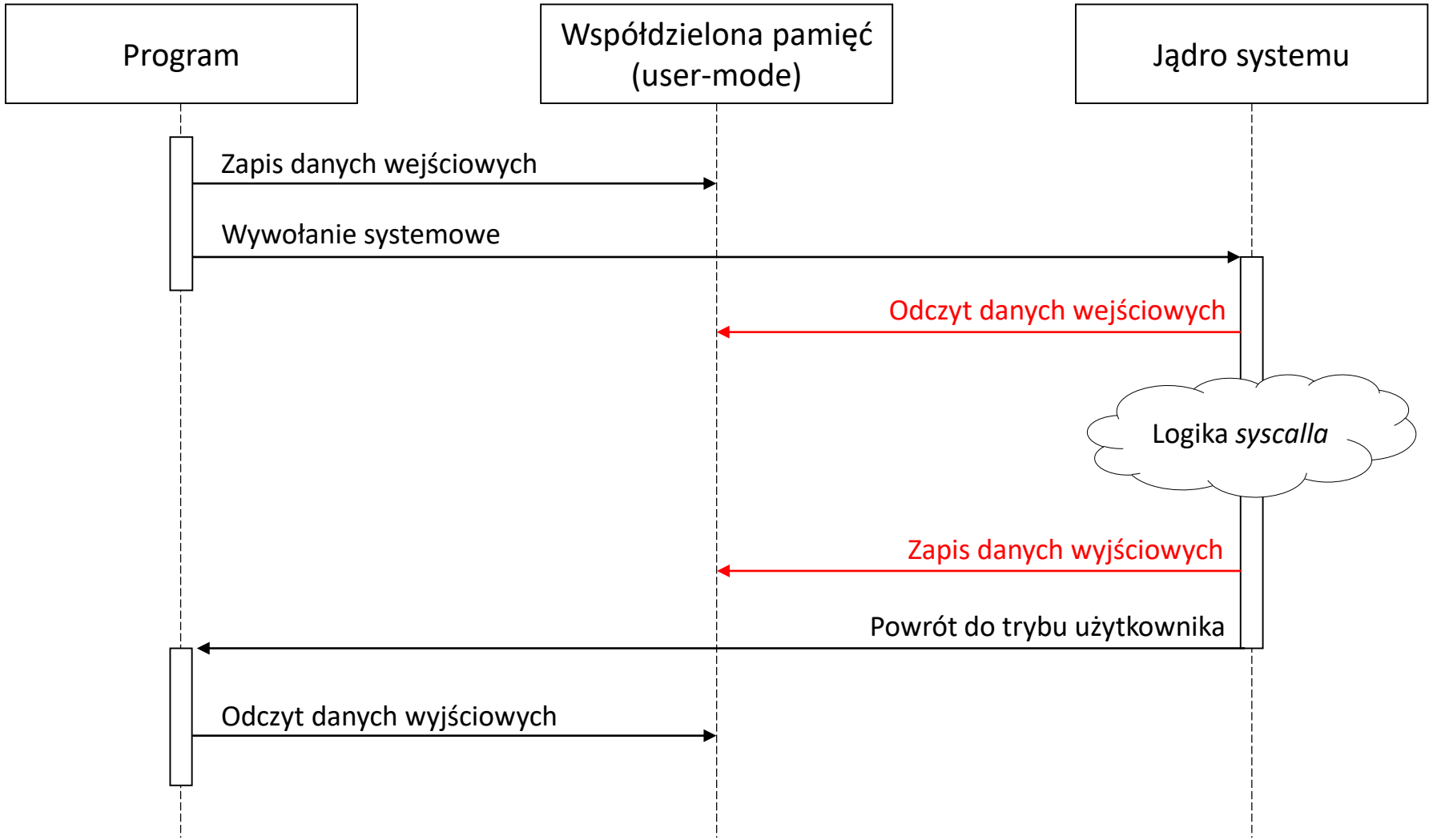
Budowa systemów operacyjnych

- Programy trybu użytkownika działają niezależnie od siebie wzajemnie i jądra.
- Kiedy chcą wykonać jakąś operację w systemie, używają wywołania systemowego (*system call*).
- Kanałem wymiany danych jest współdzielona pamięć ring-3.
  - Do pewnego stopnia również rejestry procesora.

Życie wywołania systemowego



Życie wywołania systemowego



W idealnym świecie

- W obrębie jednego wywołania systemowego, każda komórka pamięci jest:
  1. Odczytana co najwyżej raz, w bezpieczny sposób

     ... a następnie ...

  2. Nadpisana co najwyżej raz, w bezpieczny sposób, wyłącznie danymi przeznaczonymi dla trybu użytkownika.

W rzeczywistości

Fotografia autorstwa Arashi Coldwind
Badania wspólnie z Gynvaelem Coldwindem

1. Odczytana ~~co najwyżej raz~~, w bezpieczny sposób

   ... a następnie ...

2. Nadpisana co najwyżej raz, w bezpieczny sposób, wyłącznie danymi przeznaczonymi dla trybu użytkownika.

W rzeczywistości

1. Odczytana co najwyżej raz, ~~**w bezpieczny sposób**~~

   ... a następnie ...

2. Nadpisana co najwyżej raz, ~~**w bezpieczny sposób**~~, wyłącznie danymi przeznaczonymi dla trybu użytkownika.

W rzeczywistości

1. Odczytana co najwyżej raz, w bezpieczny sposób

~~**... a następnie ...**~~

2. Nadpisana co najwyżej raz, w bezpieczny sposób, wyłącznie danymi przeznaczonymi dla trybu użytkownika.

W rzeczywistości

1. Odczytana co najwyżej raz, w bezpieczny sposób

... a następnie ...

2. Nadpisana ~~**co najwyżej raz**~~, w bezpieczny sposób, wyłącznie danymi przeznaczonymi dla trybu użytkownika.

Dzisiejszy temat

1. Odczytana co najwyżej raz, w bezpieczny sposób

... a następnie ...

2. Nadpisana co najwyżej raz, w bezpieczny sposób, ~~**wyłącznie danymi przeznaczonymi dla trybu użytkownika**~~.

API między user i kernel-mode

- Istotne różnice w stosunku do wysokopoziomowych interfejsów.
  - *Serwer* (jądro) zaimplementowane w natywnych językach C, C++, assembler.
  - Informacje przekazywane przez typowe obiekty tych języków: tablice, struktury, unie itd.
  - Pamięć zaalokowana na stosie i stercie z reguły nie jest zerowana.
    - Zawiera “śmieci” znajdujące się w tym miejscu wcześniej.
  - Zakładamy możliwość wykonania kodu na atakowanym systemie, więc mamy bezpośredni dostęp do wszystkich bajtów zwróconych przez jądro.

Łatwy problem – typy podstawowe

```
NTSTATUS NtMultiplyByTwo(DWORD InputValue, LPDWORD OutputPointer) {
  DWORD OutputValue;

  if (InputValue != 0) {
    OutputValue = InputValue * 2;
  }

  *OutputPointer = OutputValue;
  return STATUS_SUCCESS;
}
```

Niezainicjalizowane jeśli
InputValue == 0

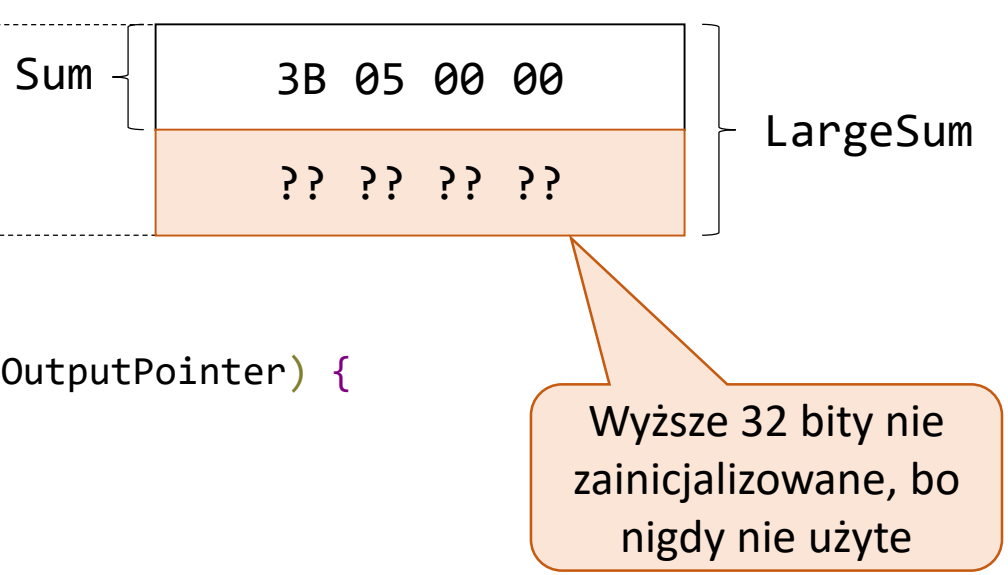
Trudny problem – struktury i unie

```
typedef struct _SYSCALL_OUTPUT {
  DWORD Sum;
  DWORD Product;
  DWORD Reserved;
} SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;

NTSTATUS NtArithOperations(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
  SYSCALL_OUTPUT OutputStruct;

  OutputStruct.Sum = InputValue + 2;
  OutputStruct.Product = InputValue * 2;

  RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));
  return STATUS_SUCCESS;
}
```

Nigdy nie zainicjalizowane, bo „zarezerwowane”

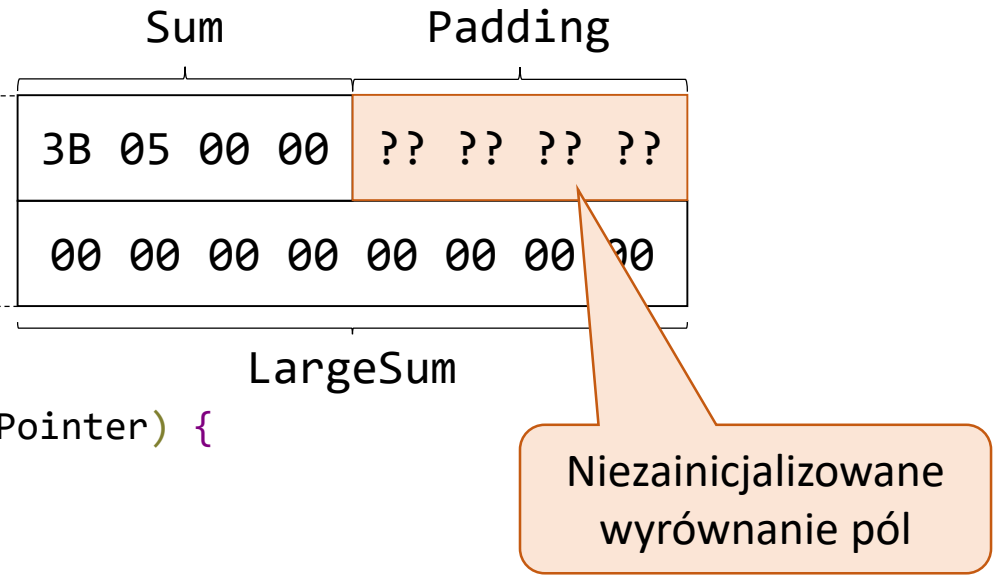
Trudny problem – struktury i unie

```
typedef union _SYSCALL_OUTPUT {
  DWORD Sum;
  QWORD LargeSum;
} SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;

NTSTATUS NtSmallSum(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
  SYSCALL_OUTPUT OutputUnion;

  OutputUnion.Sum = InputValue + 2;

  RtlCopyMemory(OutputPointer, &OutputUnion, sizeof(SYSCALL_OUTPUT));
  return STATUS_SUCCESS;
}
```

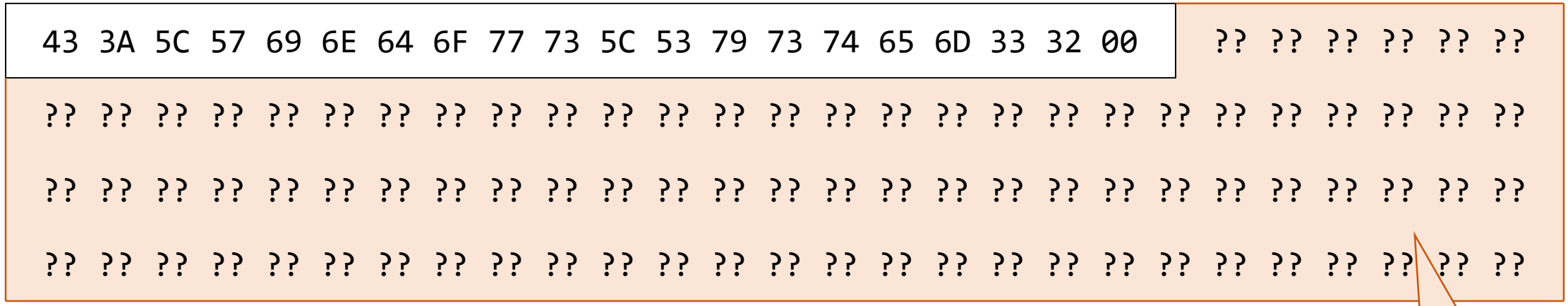


Trudny problem – struktury i unie

```
typedef struct _SYSCALL_OUTPUT {
  DWORD Sum;
  QWORD LargeSum;
} SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;

NTSTATUS NtSmallSum(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
  SYSCALL_OUTPUT OutputStruct;

  OutputStruct.Sum = InputValue + 2;
  OutputStruct.LargeSum = 0;

  RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));
  return STATUS_SUCCESS;
}
```



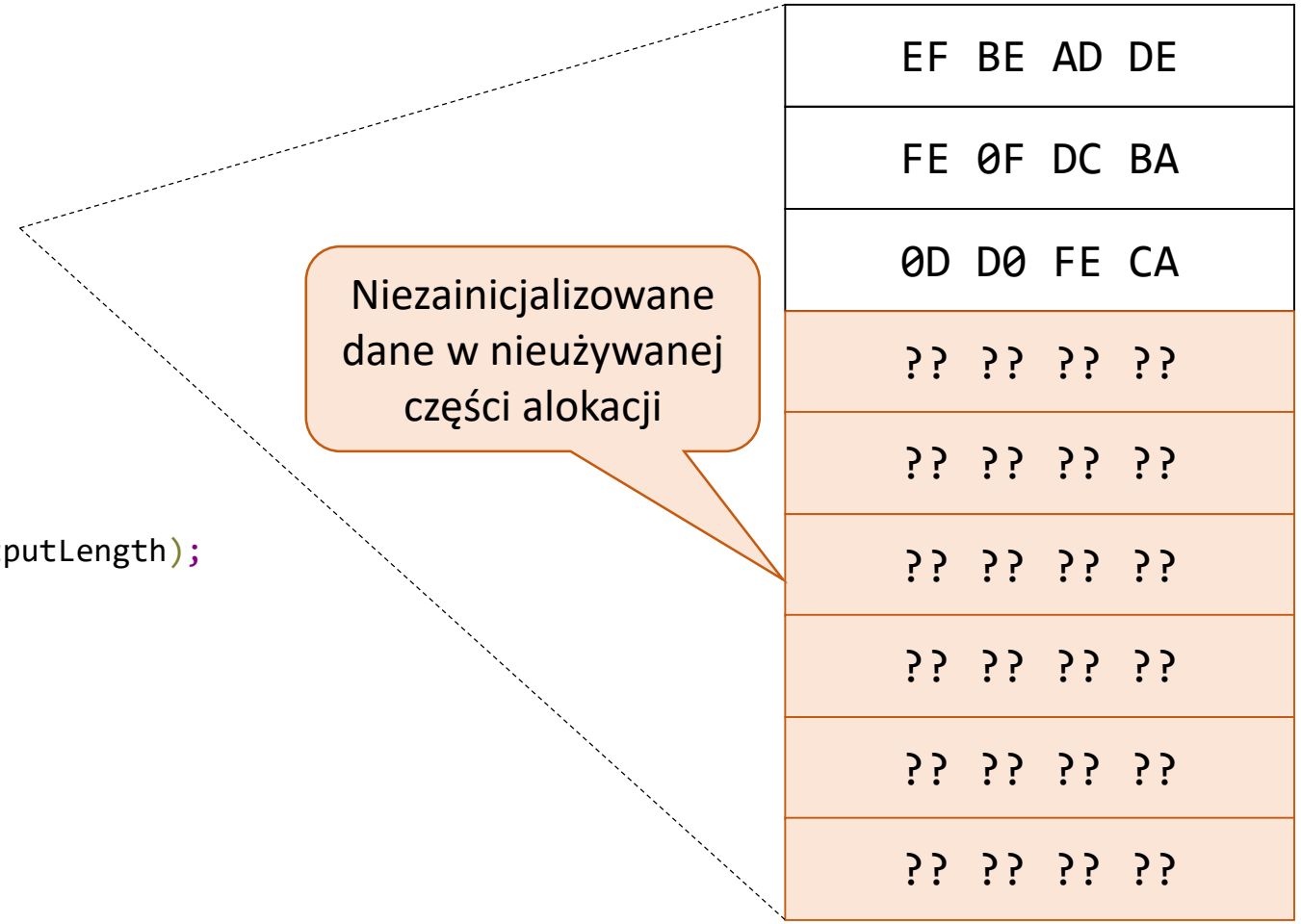
Trudny problem – tablice o stałym rozmiarze




```
NTSTATUS NtGetSystemPath(PCHAR OutputPath) {
  CHAR SystemPath[MAX_PATH] = "C:\\Windows\\System32";

  RtlCopyMemory(OutputPath, SystemPath, sizeof(SystemPath));
  return STATUS_SUCCESS;
}
```

Trudny problem – arbitralne długości zapytań

```
NTSTATUS NtMagicValues(LPDWORD OutputPointer, DWORD OutputLength) {
  if (OutputLength < 3 * sizeof(DWORD)) {
    return STATUS_BUFFER_TOO_SMALL;
  }

  LPDWORD KernelBuffer = Allocate(OutputLength);

  KernelBuffer[0] = 0xdeadbeef;
  KernelBuffer[1] = 0xbadc0ffe;
  KernelBuffer[2] = 0xcafed00d;

  RtlCopyMemory(OutputPointer, KernelBuffer, OutputLength);
  Free(KernelBuffer);

  return STATUS_SUCCESS;
}
```



Dodatkowo: brak automatycznej inicjalizacji

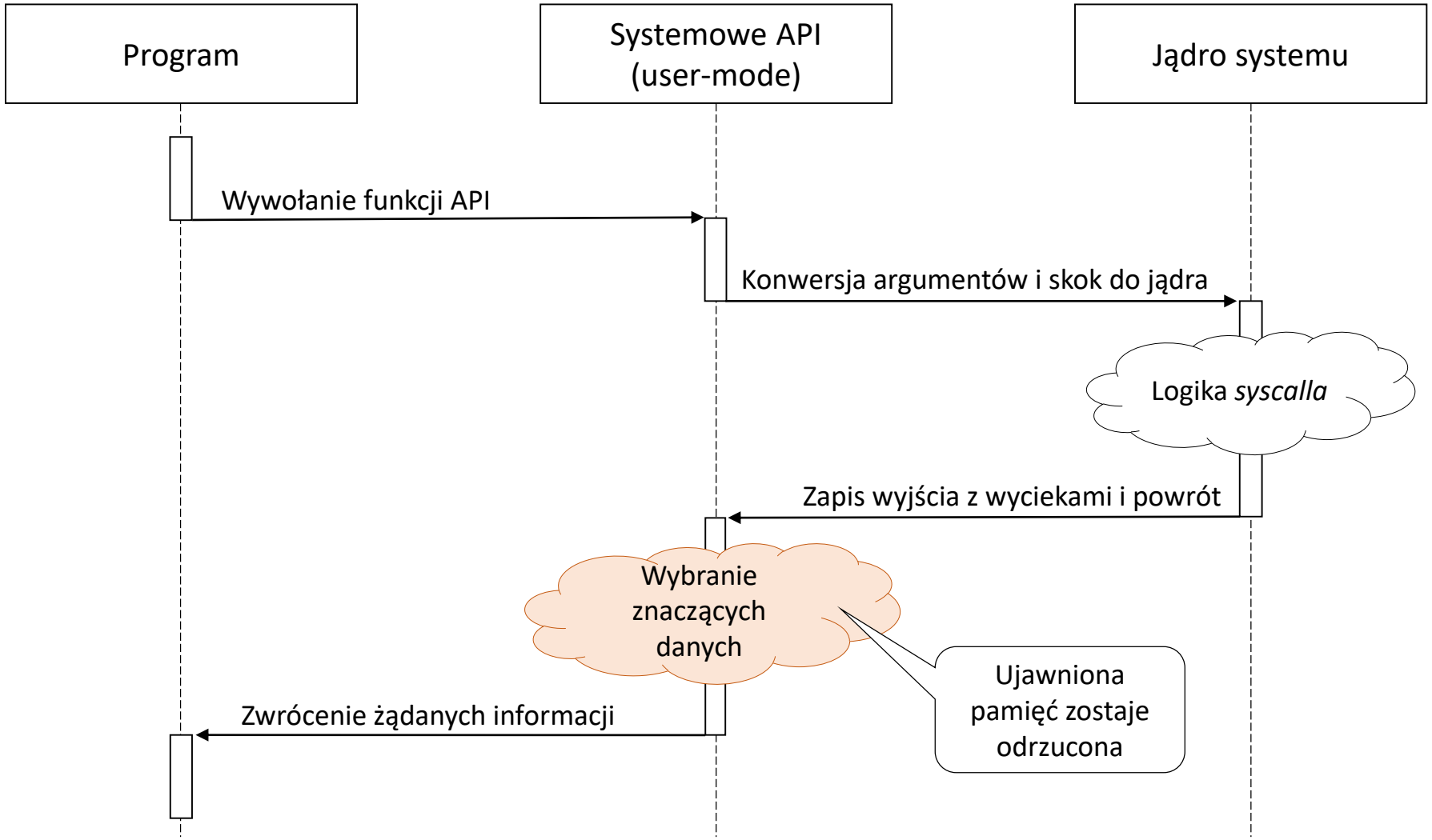
- Z reguły Windows ani Linux nie inicjalizują nowych obiektów ze stosu/sterty.
  - Z pewnymi wyjątkami, głównie na Linuksie: **`kzalloc()`**, **`__GFP_ZERO`**, **`PAX_MEMORY_STACKLEAK`** itp.
  - Bufor tzw. *buffered IOCTL* jest czyszczony w Windowsie od czerwca 2017 (**nowość!**)
- Z MSDN:

> **Note** Memory that **ExAllocatePoolWithTag** allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).

Dodatkowo: brak konsekwencji

- C/C++ nie są zaprojektowane do bezpiecznego przesyłania danych pomiędzy domenami bezpieczeństwa, ale robienie tego w sposób niebezpieczny nie niesie widocznego ryzyka.
  - Nawet jeśli jądro ujawni kilka niezainicjalizowanych bajtów tu i tam, system będzie działał i nikt nigdy tego nie zauważy (do teraz ☺).
- Jeśli programista jądra nie jest świadomy tej klasy błędów i nie próbuje jej świadomie zapobiegać, to prawdopodobnie nigdy nie zauważy jej przez przypadek.

Dodatkowo: wycieki ukryte za API



Wykorzystanie wycieków danych

- „Tylko” lokalne ujawnienia danych, z definicji brak możliwości uruchomienia kodu lub zdalnej exploitacji.
- Poziom zagrożenia zależy od tego, co konkretnie może wyciec z jądra.
- Na “plus”, większość wycieków nie pozostawia po sobie żadnych śladów, więc można próbować w nieskończoność bez obawy o wykrycie i stabilność systemu.

Wykorzystanie wycieków danych

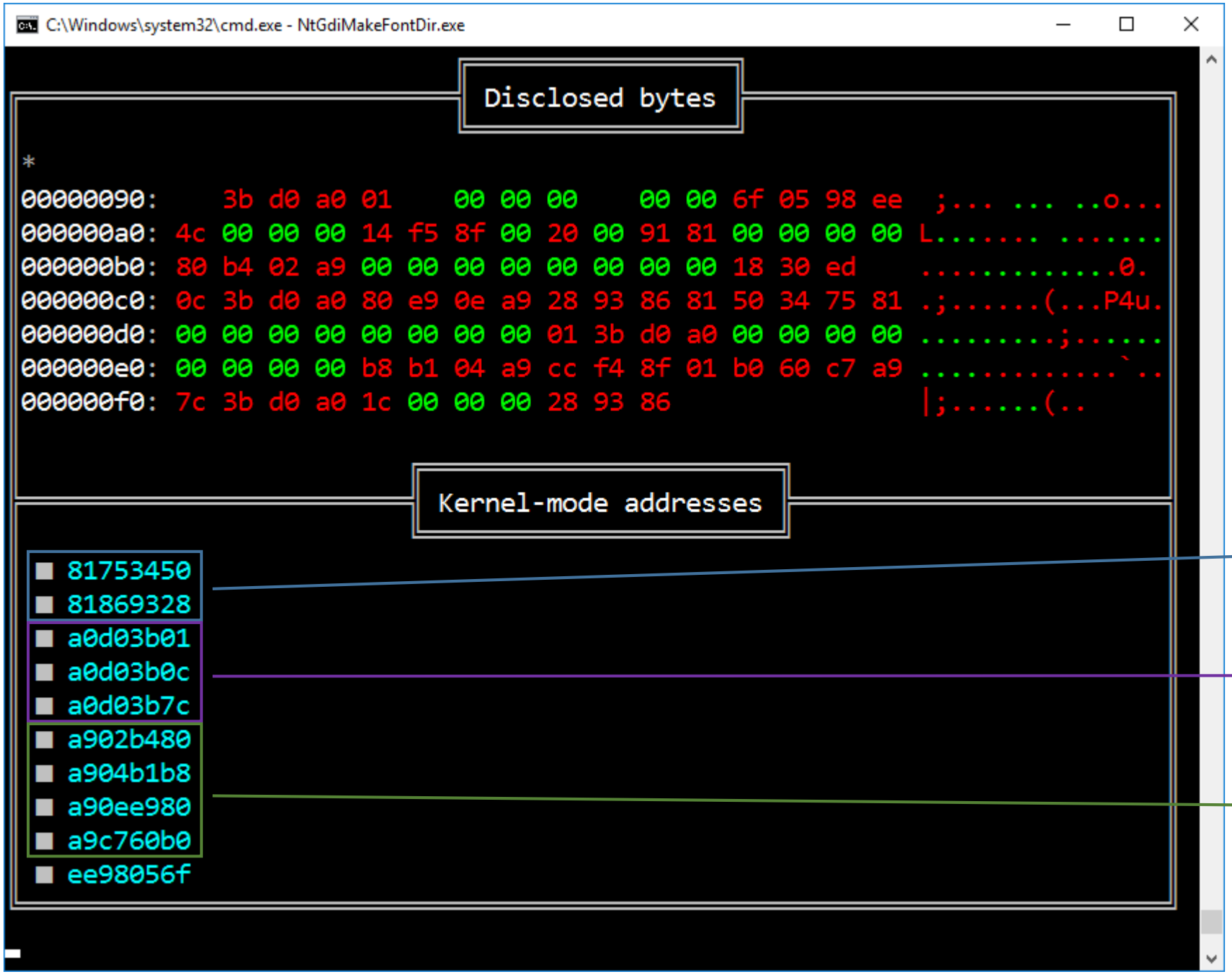
- Szczególnie przydatne jako część exploitów podniesienia uprawnień w systemie.
  - Zwłaszcza biorąc pod uwagę, jak dużo wysiłku poświęca się KASLR i ochronie informacji o układzie przestrzeni adresowej trybu jądra.
- Przykład z życia: exploit na jądro Windows znaleziony w archiwach Hacking Team w lipcu 2015 (CVE-2015-2433, MS15-080).
  - Wyciek pamięci sterty ujawniający adres sterownika systemowego win32k.sys.
  - Niezależnie odkryty przez Matta Taita z P0, Issue #480.

Kernel-mode ASLR leak via uninitialized memory returned to usermode by NtGdiGetTextMetrics

Reported by matttait@google.com, Jul 10 2015

Przydatność wycieków ze stosu

- Mało różnorodne dane, użyteczne głównie w exploitacji innych podatności.
  - Adresy stosu, sterty i sterowników jądra.
  - Wartości “ciasteczek” chroniących przed przepełnieniami bufora.
  - Dane zapisane przez wywołania systemowe używane wcześniej przez ten sam wątek.



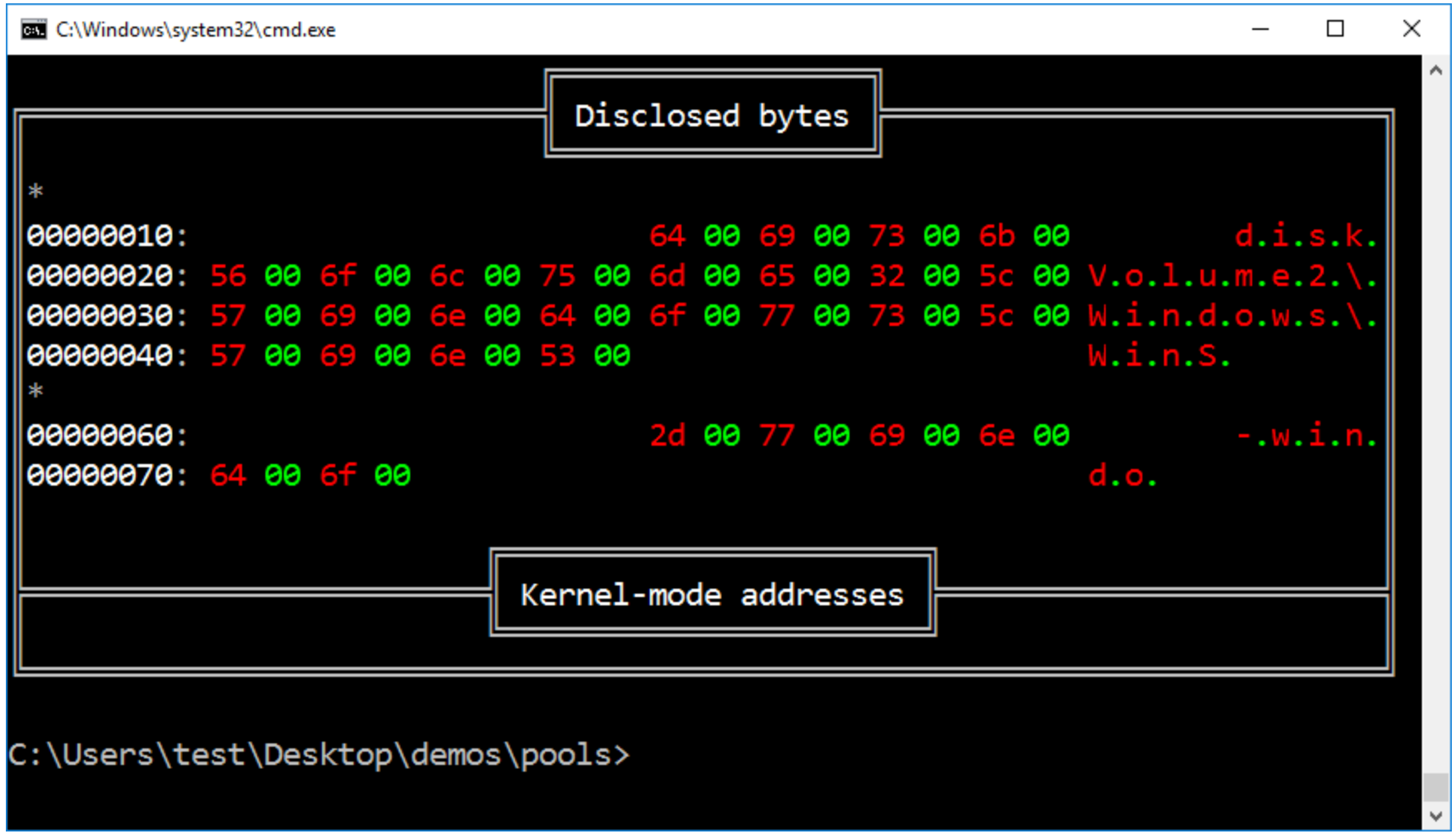
Adresy kodu jądra (ntoskrnl.exe)

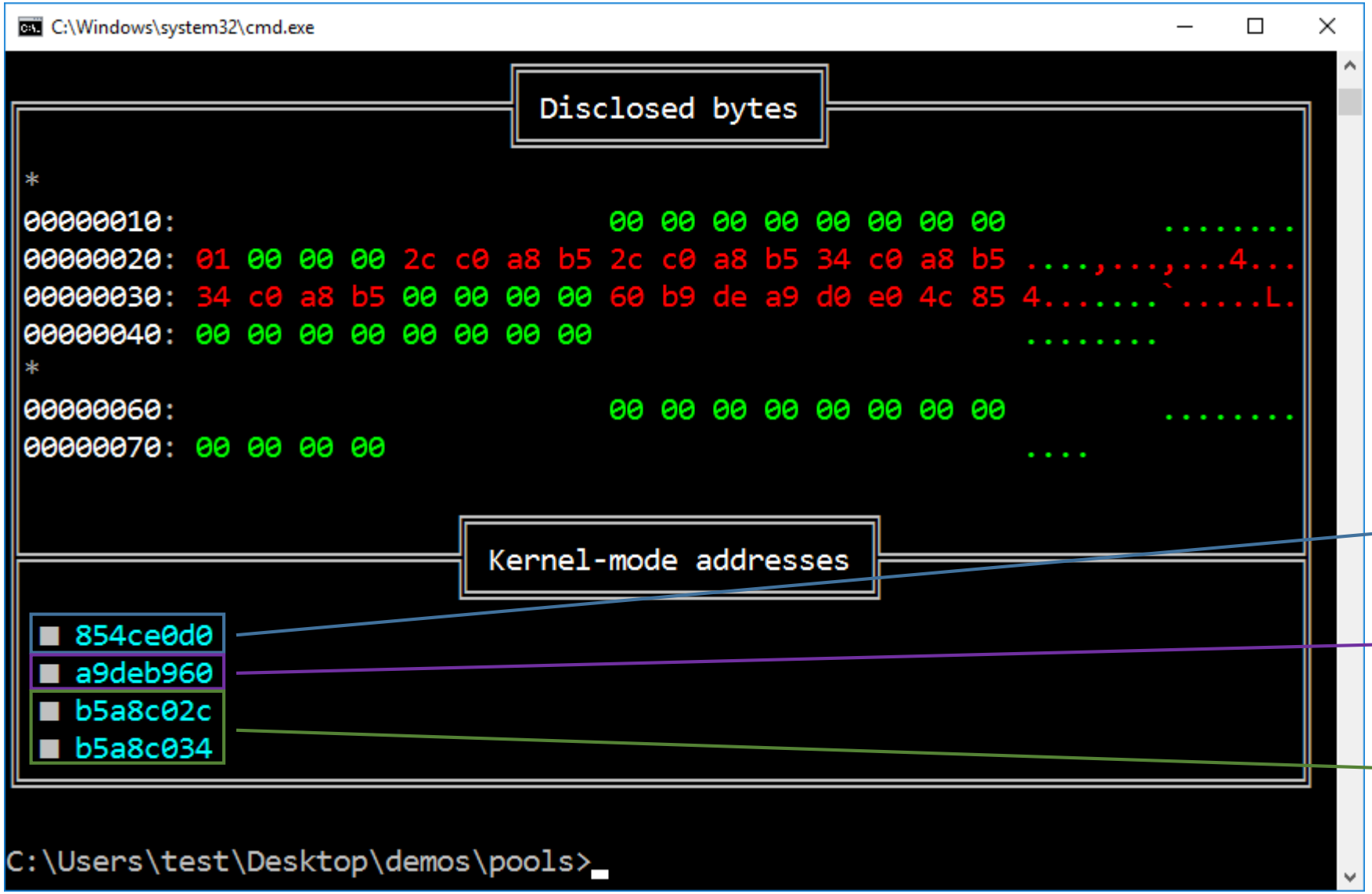
Adresy stosu jądra

Adresy sterty jądra (*non-paged pool*)

Przydatność ujawnień ze sterty

- Mniej *oczywiste* dane, ale z większym potencjałem do zdradzenia rozmaitych informacji przetwarzanych przez jądro:
  - Adresy alokacji na stercie i modułów wykonywalnych w pamięci.
  - Teoretycznie dane dowolnego sterownika – dysku, sieciowego, video, urządzeń peryferyjnych itd.
    - W zależności od typu sterty, rozmiaru alokacji i aktywności w systemie w danym momencie.


C:\Windows\system32\cmd.exe
Disclosed bytes
*
00000010: 64 00 69 00 73 00 6b 00 d.i.s.k.
00000020: 56 00 6f 00 6c 00 75 00 6d 00 65 00 32 00 5c 00 V.o.l.u.m.e.2.\.
00000030: 57 00 69 00 6e 00 64 00 6f 00 77 00 73 00 5c 00 W.i.n.d.o.w.s.\.
00000040: 57 00 69 00 6e 00 53 00 W.i.n.S.
*
00000060: 2d 00 77 00 69 00 6e 00 -.w.i.n.
00000070: 64 00 6f 00 d.o.
Kernel-mode addresses
C:\Users\test\Desktop\demos\pools>



Adresy sterty:

Duża alokacja, *non-paged*

Mała alokacja, *non-paged*

Małe alokacje, *paged*

Dotychczasowe badania (Windows)

- Do 2017 r. problem praktycznie nieporuszany, oprócz kilku wyjątków:
  - Wyciek w `NtGdiGetTextMetrics` (CVE-2015-2433) w 2015 – Hacking Team i Matt Tait
  - 8 wycieków w sterowniku win32k.sys w 2015 – Wandering Glitch
- W tym roku temat zdobył nieco większe zainteresowanie.
  - Wzmianki podczas prelekcji na CanSecWest – Peng Qiu i SheFang Zhong
  - Konkurencyjne badania *Automatically Discovering Windows Kernel Information Leak Vulnerabilities* – fanxiaocao and pjf of IceSword Lab (Qihoo 360)
  - Pojedyncze błędy zgłaszane przez innych badaczy.

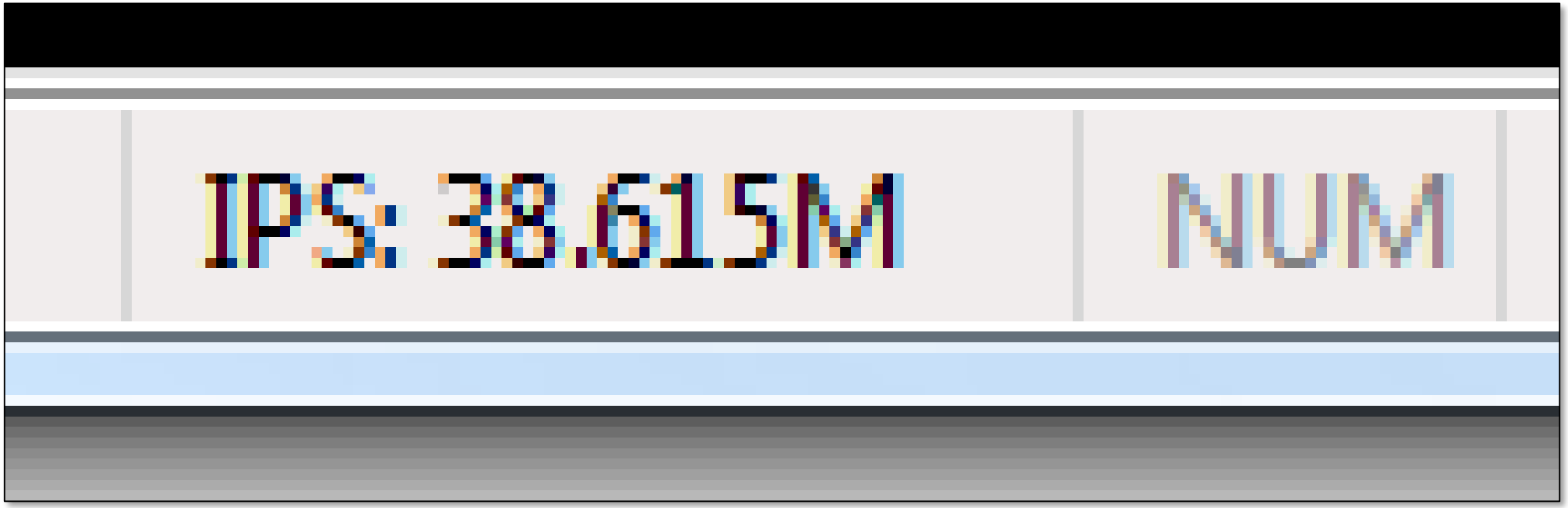
Dotychczasowe badania (Linux)

- W 2010 r. Dan Rosenberg zgłosił ponad 20 ujawnień pamięci w różnych podsystemach jądra.
  - Część badań opisał podczas prelekcji *Stackjacking and Other Kernel Nonsense*, zaprezentowanej przez niego i Jona Oberheide w 2011 r.
- Duża ilość łatek została nadesłana przez lata przez różnych badaczy, np.: Salva Peiró, Clément Lecigne, Marcel Holtmann, Kees Cook, Jeff Mahoney itd.
- Wygląda na to, że ten rodzaj błędów jest dość dobrze znany programistom Linuksa.

Projekt Bochspwn Reloaded

- Bochs jest pełnym emulatorem architektury IA-32 i AMD64.
  - CPU i wszystkie podstawowe urządzenia – cały wyemulowany komputer.
- Napisany w C++.
- Wspiera najnowsze procesory i ich rozszerzenia.
  - SSE, SSE2, SSE3, SSSE3, SSE4, AVX, AVX2, AVX512, SVM / VT-x etc.
- Poprawnie radzi sobie z popularnymi systemami operacyjnymi.
- Posiada rozbudowane API do instrumentacji emulowanego kodu.

Wydajność (w skrócie)



Wydajność (dłuższa historia)

- Na współczesnych maszynach system-gość działa z prędkością **80-100M IPS**.
  - Wystarcza do uruchomienia systemu w rozsądnym czasie (5-10 minut).
  - Środowisko dość responsywne, GUI wyświetla ok. 1-5 klatek na sekundę.
- Instrumentacja dodaje pewien narzut.
  - W przypadku *Bochspwn Reloaded* wydajność spada do **30-40M IPS**.
  - Prosta logika i szybka implementacja to klucz do sukcesu.

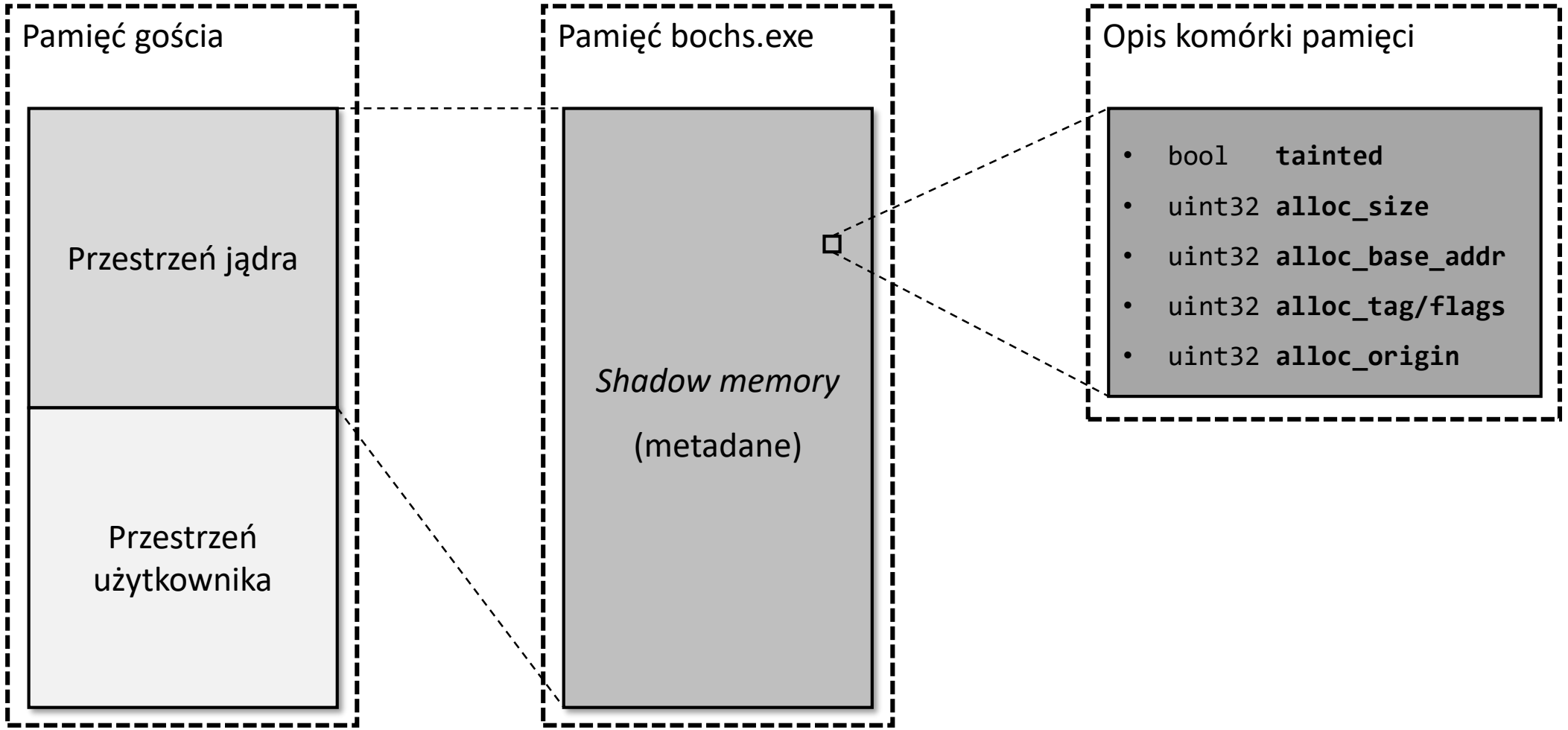
Zasadniczy pomysł

- Implementacja tzw. *taint trackingu* dla całej przestrzeni adresowej jądra.
- Podstawowa funkcjonalność:
    1. *Oznaczanie* nowych alokacji na stosie i stercie.
    2. Usuwanie oznaczenia (*taintu*) podczas zwalniania pamięci na stercie.
    3. Propagacja oznaczenia pamięci podczas kopiowania danych.
    4. Wykrywanie przekazywania oznaczonej pamięci do trybu użytkownika.

Pomocnicze funkcje

- Śledzenie listy aktualnie załadowanych modułów jądra.
- Odczytywanie stosu wywołań w momencie wykrycia błędu (w celu deduplikacji).
- Symbolizacja adresów zapisywanych w raportach tekstowych.
- Przekazanie kontroli debuggerowi jądra w momencie wykrycia błędu.

Reprezentacja *shadow memory*



Reprezentacja *shadow memory*

- Liniowa w stosunku do rozmiaru przestrzeni adresowej jądra w systemie-gościu.
  - W tym momencie wspierane wyłącznie 32-bitowe systemy.
  - Niektóre informacje przechowujemy z dokładnością do 1 bajtu, inne dla każdych 8 bajtów.
- Zawiera dodatkowe informacje przydatne w momencie zgłaszania błędów.
- Maksymalne zużycie pamięci:
  - Windows (2 GB przestrzeni jądra) – **6 GB**
  - Linux (1 GB przestrzeni jądra) – **3 GB**
  - Bezproblemowy narzut przy wystarczającej ilości RAMu na systemie-hoście.

Oznaczanie alokacji na stosie

- Łatwe, uniwersalne, niezależne od systemu.
- Wykrycie instrukcji modyfikującej rejestr ESP:

```
ADD ESP, ...        SUB ESP, ...        AND ESP, ...
```

- Po wykonaniu, jeśli ESP zostało zmniejszone, wywołujemy:

$$\texttt{set\_taint}(\texttt{ESP}_{\texttt{old}}, \texttt{ESP}_{\texttt{new}})$$

Oznaczanie alokacji na stercie

- Specyficzne dla konkretnego systemu.
- Wymaga znajomości zarówno zwróconego adresu alokacji, jak i argumentów (rozmiaru, flag itd.) w tym samym momencie.
- Wtedy:

```
set_taint(address, address + size)
```

Propagacja taintu

- Najtrudniejsza część – wykrywanie kopiowania danych.
- Bochspwn propaguje taint dla instrukcji `<REP> MOVS{B,D}`.
  - Używana przez różne formy `memcpy()`.
  - Zarówno adres źródłowy (`ESI`) jak i docelowy (`EDI`) są znane w tym samym czasie.
  - I tak interesuje nas głównie kopiowanie pełnych obszarów pamięci (a nie pojedynczych zmiennych).
- Jeśli zapis do pamięci nie jest skutkiem `<REP> MOVS{B,D}`, oznaczamy bajty jako zainicjalizowane.

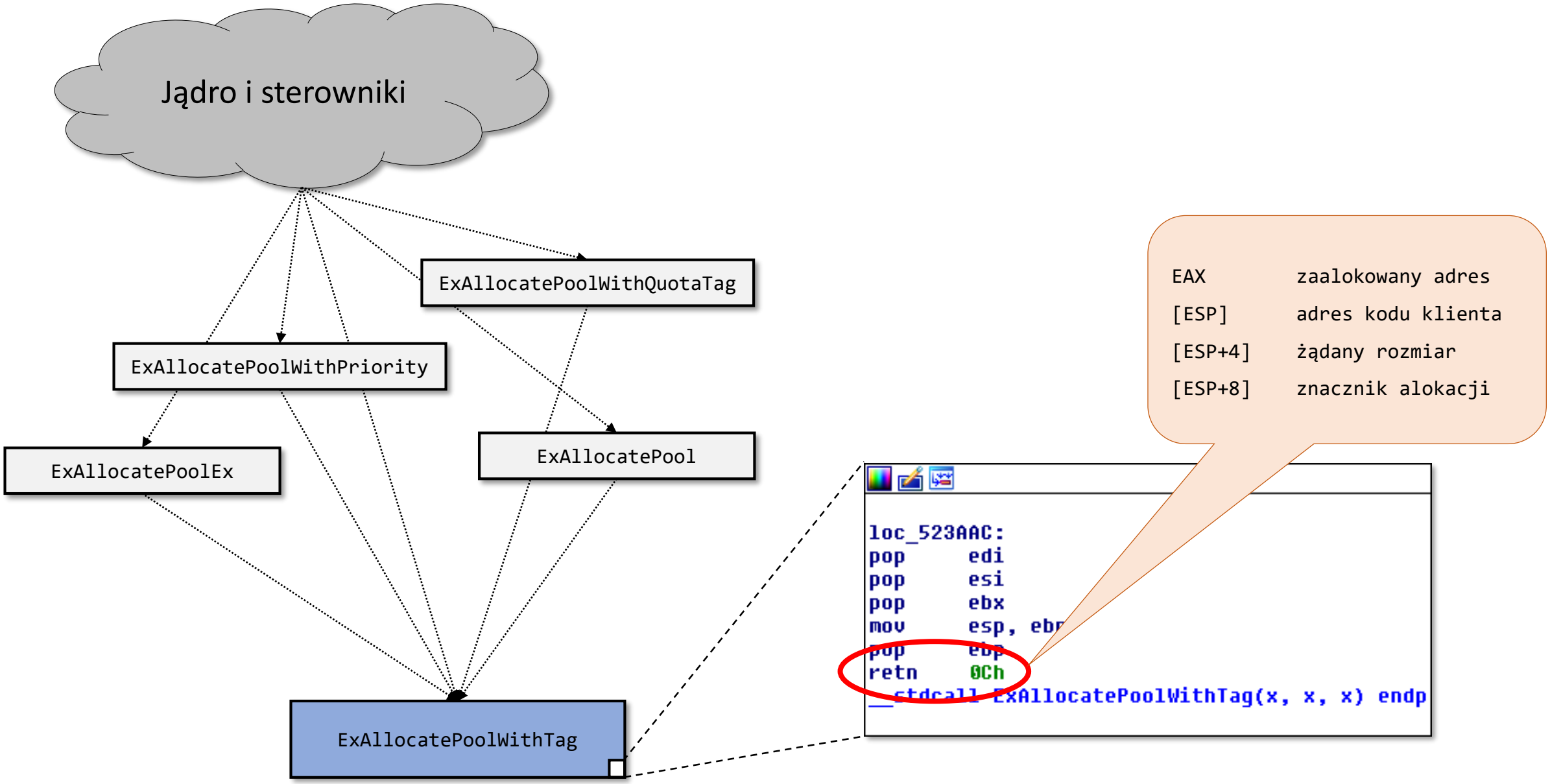
Wykrywanie błędów

- Aktywowane dla instrukcji `<REP> MOVS{B,D}` gdzie adres źródłowy jest w przestrzeni jądra, a docelowy – użytkownika.
  - Sprawdzamy taint całego kopiowanego regionu.
  - Jeśli co najmniej jeden bajt jest niezainicjalizowany, zgłaszamy błąd.

Bochspwn vs. Windows

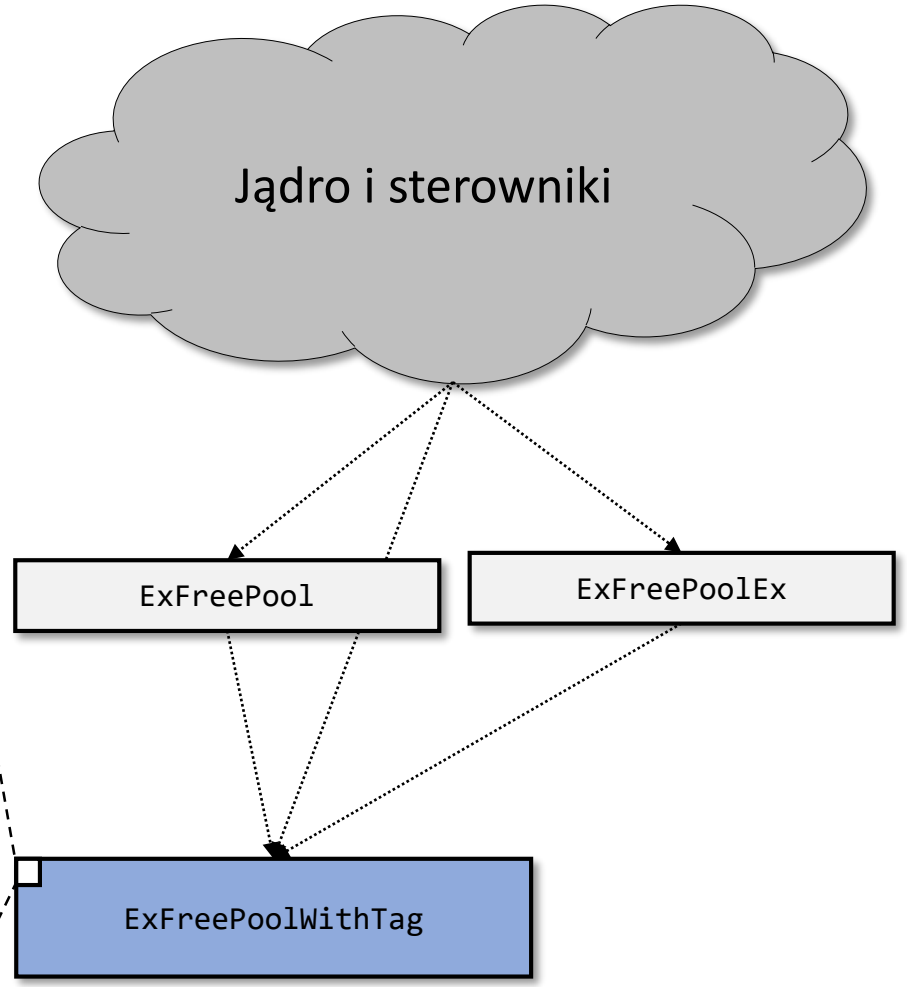
Szczegóły implementacyjne

- Alokator sterty sprowadza się do dwóch funkcji: **`ExAllocatePoolWithTag`** i **`ExFreePoolWithTag`**.
- Kopiowanie pamięci odbywa się przez **`rep movs`**.
- Drobne wyjątki od powyższych reguł:
  - Zoptymalizowane alokatory, np. `win32k!AllocFreeTmpBuffer`.
  - Zoptymalizowana implementacja `memcpy()` dla długości < 32.

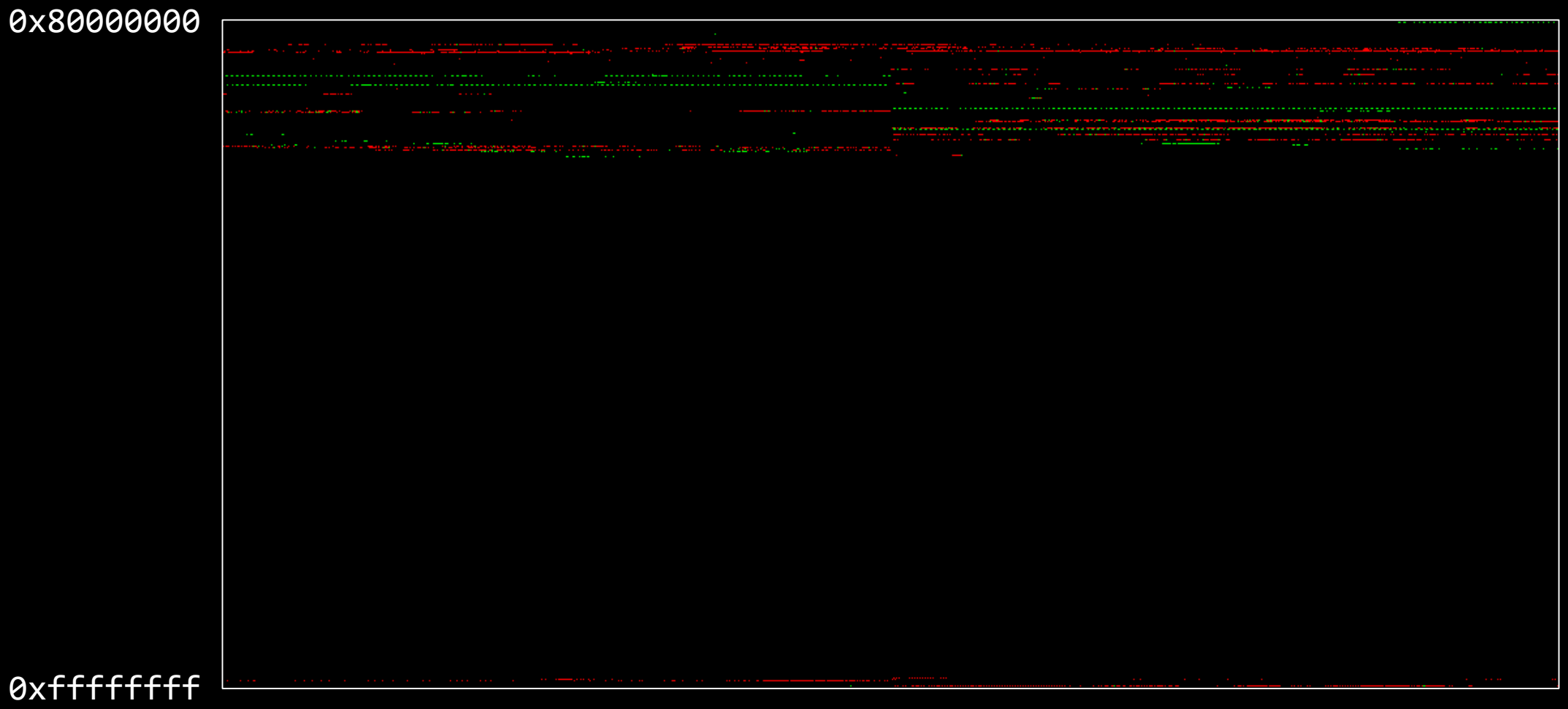

Jądro i sterowniki
ExAllocatePoolWithQuotaTag
ExAllocatePoolWithPriority
ExAllocatePoolEx
ExAllocatePool
ExAllocatePoolWithTag
EAX zaalokowany adres
[ESP] adres kodu klienta
[ESP+4] żądany rozmiar
[ESP+8] znacznik alokacji
loc_523AAC:
pop edi
pop esi
pop ebx
mov esp, ebp
pop ebp
retn 0Ch
__stdcall ExAllocatePoolWithTag(x, x, x) endp

```
; Exported entry 229. ExFreePoolWithTag


; Attributes: bp-based frame

; void __stdcall ExFreePoolWithTag(PVOID P, ULONG Tag)
public __stdcall ExFreePoolWithTag(x, x)
__stdcall ExFreePoolWithTag(x, x) proc near

var_48= dword ptr -48h
var_44= dword ptr -44h
var_40= dword ptr -40h
var_3C= dword ptr -3Ch
var_38= dword ptr -38h
var_34= dword ptr -34h
var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
LockHandle= _KLOCK_QUEUE_HANDLE ptr -0Ch
P= dword ptr  8
Tag= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF8h
mov     eax, _ExpSpecialAllocations
sub     esp, 4Ch
push    ebx
push    esi
mov     esi, [ebp+P]
push    edi
test    eax, eax
jz      loc_523B95
```

[ESP+4] zwalniany adres



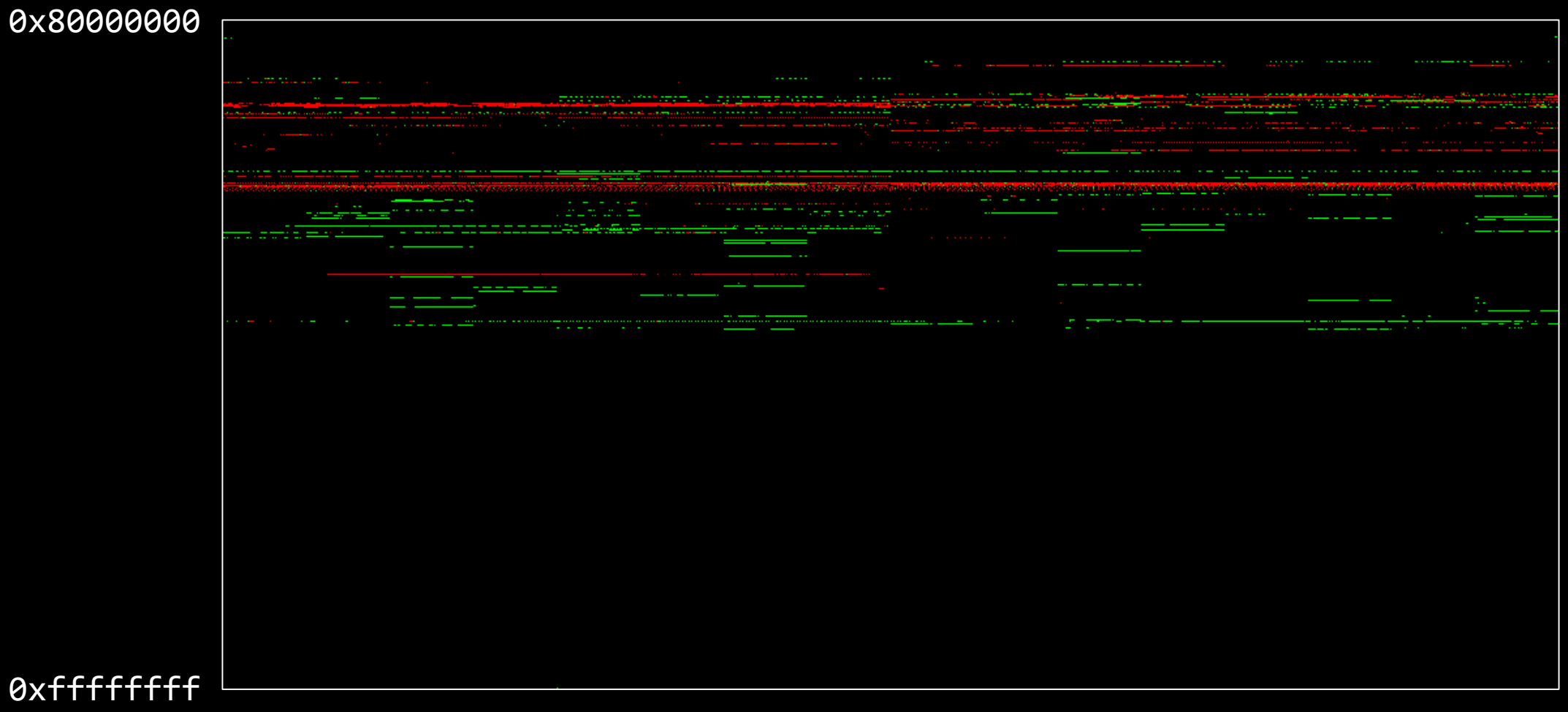
Układ pamięci jądra Windows 7



strony stosu strony sterty

40 minut działania, interwał 20s., uruchomienie + początkowe testy ReactOS

Układ pamięci jądra Windows 10



strony stosu strony sterty

120 minut działania, interwał 60s., uruchomienie + początkowe testy ReactOS

Przykładowy raport błędu

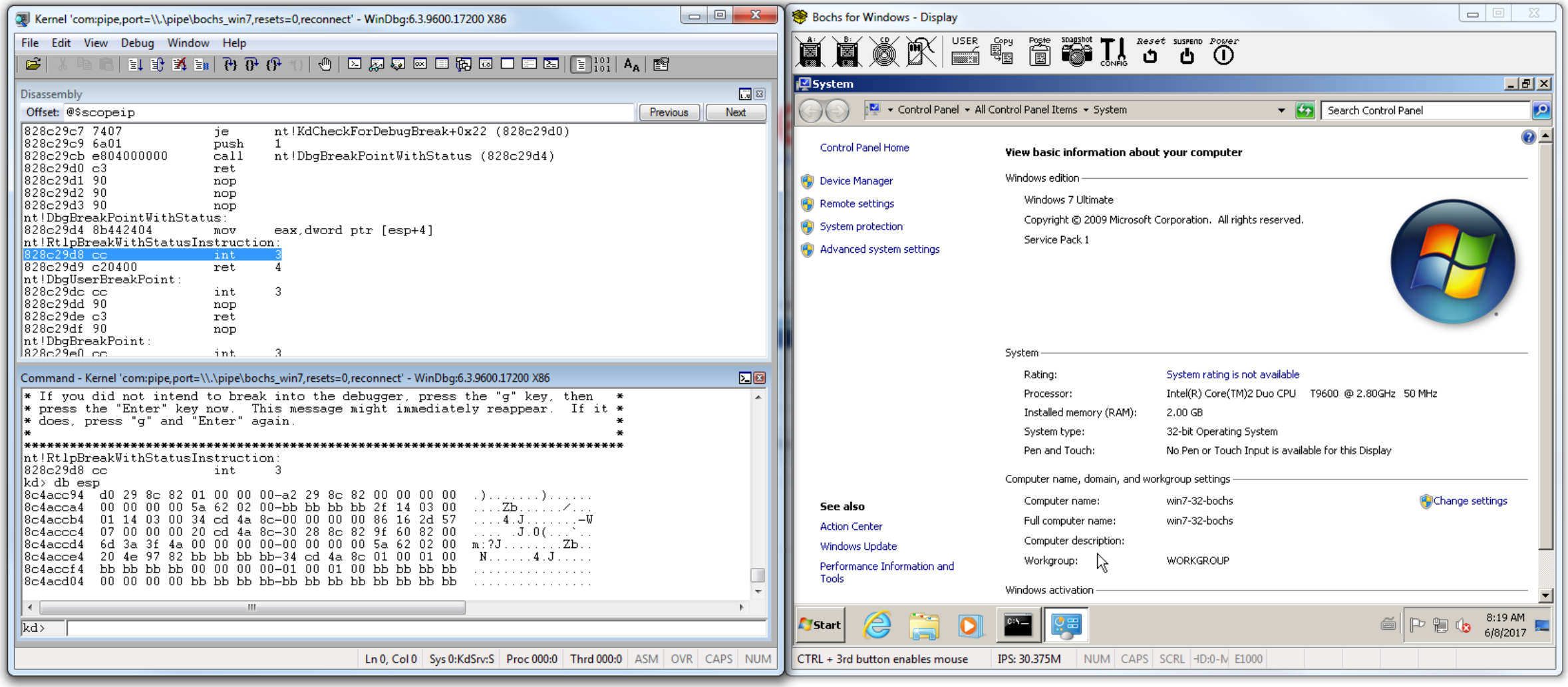
```
------------------------------ found uninit-access of address 94447d04
[pid/tid: 000006f0/00000740] {    explorer.exe}
       READ of 94447d04 (4 bytes, kernel--->user), pc = 902df30f
       [ rep movsd dword ptr es:[edi], dword ptr ds:[esi] ]
[Pool allocation not recognized]
Allocation origin: 0x90334988 ((000c4988) win32k.sys!__SEH_prolog4+00000018)
Destination address: 1b9d380
Shadow bytes: 00 ff ff ff Guest bytes: 00 bb bb bb
Stack trace:
 #0  0x902df30f ((0006f30f) win32k.sys!NtGdiGetRealizationInfo+0000005e)
 #1  0x8288cdb6 ((0003ddb6) ntoskrnl.exe!KiSystemServicePostCall+00000000)
```

Wsparcie dla debuggera jądra

- Tekstowe raporty Bochspwna są dość szczegółowe, ale nie zawsze wystarczające, by znaleźć i zreprodukować błąd.
  - Zwłaszcza dla IOCTL i innych skomplikowanych przypadków, gdzie trzeba wgłębić się w kontekst wykonania i stan systemu.
- Rozwiązanie – podpiąć WinDbg do jądra emulowanego systemu!
  - Łatwa konfiguracja, Bochs pozwala na przekierowanie portu COM na Windowsowe *pipe'y*.
  - Oczywiście wolne, jak wszystko pod Bochsem, ale da się z tym pracować. ☺

Analiza w momencie wycieku

- Podpięty debugger jest mało przydatny, jeśli nie możemy debugować systemu dokładnie w momencie ujawnienia pamięci.
- Stąd – po zapisaniu raportu do pliku, Bochspwn generuje wyjątek INT3 w emulatorze.
  - WinDbg zatrzymuje się zaraz po instrukcji `rep movs`, która ujawniła dane.
- Wyjątek generowany przez dodatkową instrumentację w emulatorze x86, który zatrzymuje podpięty do emulowanego jądra debugger – magia. ☺


Kernel 'com:pipe,port=\\.\pipe\bochs_win7,resets=0,reconnect' - WinDbg:6.3.9600.17200 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
Previous
Next
828c29c7 7407 je nt!KdCheckForDebugBreak+0x22 (828c29d0)
828c29c9 6a01 push 1
828c29cb e804000000 call nt!DbgBreakPointWithStatus (828c29d4)
828c29d0 c3 ret
828c29d1 90 nop
828c29d2 90 nop
828c29d3 90 nop
nt!DbgBreakPointWithStatus:
828c29d4 8b442404 mov eax,dword ptr [esp+4]
nt!RtlpBreakWithStatusInstruction:
828c29d8 cc int 3
828c29d9 c20400 ret 4
nt!DbgUserBreakPoint:
828c29dc cc int 3
828c29dd 90 nop
828c29de c3 ret
828c29df 90 nop
nt!DbgBreakPoint:
Command - Kernel 'com:pipe,port=\\.\pipe\bochs_win7,resets=0,reconnect' - WinDbg:6.3.9600.17200 X86
* If you did not intend to break into the debugger, press the "g" key, then *
* press the "Enter" key now. This message might immediately reappear. If it *
* does, press "g" and "Enter" again. *
nt!RtlpBreakWithStatusInstruction:
828c29d8 cc int 3
kd> db esp
8c4acc94 d0 29 8c 82 01 00 00 00-a2 29 8c 82 00 00 00 00
8c4acca4 00 00 00 00 5a 62 02 00-bb bb bb bb 2f 14 03 00
8c4accb4 01 14 03 00 34 cd 4a 8c-00 00 00 00 86 16 2d 57
8c4accc4 07 00 00 00 20 cd 4a 8c-30 28 8c 82 9f 60 82 00
8c4accd4 6d 3a 3f 4a 00 00 00 00-00 00 00 00 5a 62 02 00
8c4acce4 20 4e 97 82 bb bb bb bb-34 cd 4a 8c 01 00 01 00
8c4accf4 bb bb bb bb 00 00 00 00-01 00 01 00 bb bb bb bb
8c4acd04 00 00 00 00 bb bb bb bb-bb bb bb bb bb bb bb bb
kd>
Ln 0, Col 0
Sys 0:KdSrv:S
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM
Bochs for Windows - Display
A:
B:
CD
USER
Copy
Paste
snapshot
CONFIG
Reset
SUSPEND
Power
System
Control Panel All Control Panel Items System
Search Control Panel
Control Panel Home
Device Manager
Remote settings
System protection
Advanced system settings
View basic information about your computer
Windows edition
Windows 7 Ultimate
Copyright © 2009 Microsoft Corporation. All rights reserved.
Service Pack 1
System
Rating: System rating is not available
Processor: Intel(R) Core(TM)2 Duo CPU T9600 @ 2.80GHz 50 MHz
Installed memory (RAM): 2.00 GB
System type: 32-bit Operating System
Pen and Touch: No Pen or Touch Input is available for this Display
Computer name, domain, and workgroup settings
Computer name: win7-32-bochs
Change settings
Full computer name: win7-32-bochs
Computer description:
Workgroup: WORKGROUP
Windows activation
See also
Action Center
Windows Update
Performance Information and Tools
Start
8:19 AM
6/8/2017
CTRL + 3rd button enables mouse
IPS: 30.375M
NUM
CAPS
SCRL
E1000

Wykonane testy

- Przetestowane systemy Windows 7 i 10.
- Przeprowadzone czynności:
  - Uruchomienie systemu.
  - Włączenie i “przeklikanie” kilku domyślnych aplikacji – **Internet Explorer**, **Wordpad**, **Edytor Rejestru**, **Panel Sterowania**, gry itp.
  - Wygenerowanie ruchu sieciowego.
  - Uruchomienie ~800 testów jednostkowych projektu **ReactOS**.
- Obszerne pokrycie kodu jądra jest wciąż głównym problemem podczas szukania błędów przy użyciu dynamicznej instrumentacji.

Wyniki

Windows Kernel Information Disclosure Vulnerability	CVE-2017-8478	Mateusz Jurczyk of Google
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8479	Mateusz Jurczyk of Google
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8480	Mateusz Jurczyk of Google
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8481	Mateusz Jurczyk of Google
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8482	• fanxiaocao and pjf of I • Mateusz Jurczyk of Go
Win32k Information Disclosure Vulnerability	CVE-2017-8484	Mateusz Jurczyk of Google
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8485	• fanxiaocao and pjf of I • Mateusz Jurczyk of Go
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8488	Mateusz Jurczyk of Google
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8489	Mateusz Jurczyk of Google
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8490	• fanxiaocao and pjf of I • Mateusz Jurczyk of Go
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8491	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8492	Mateusz Jurczyk of Google

Win32k Information Disclosure Vulnerability	CVE-2017-0245	Mateusz Jurczyk of Google
Windows Kernel Information Disclosure Vulnerability	CVE-2017-0258	Mateusz Jurczyk of Google
Windows Kernel Information Disclosure Vulnerability	CVE-2017-0259	Mateusz Jurczyk of Google

Windows Kernel Information Disclosure Vulnerability	CVE-2017-0167	• Mateusz Jurczyk of G • Meysam Firouzi (@R0

Windows Kernel Information Disclosure Vulnerability	CVE-2017-8564	Mateusz Jurczyk of Google

Volume Manager Extension Driver Information Disclosure Vulnerability	CVE-2017-8668	Mateusz Jurczyk of Google

Windows Kernel Information Disclosure Vulnerability	CVE-2017-0299	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-0300	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8462	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8469	Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8470	• fanxiaocao and pjf of IceSword Lab, Qihoo 360 • Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8471	Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8472	Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8473	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8474	• fanxiaocao and pjf of IceSword Lab , Qihoo 360 • Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8475	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8476	• fanxiaocao and pjf of IceSword Lab , Qihoo 360 • Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8477	Mateusz Jurczyk of Google Project Zero

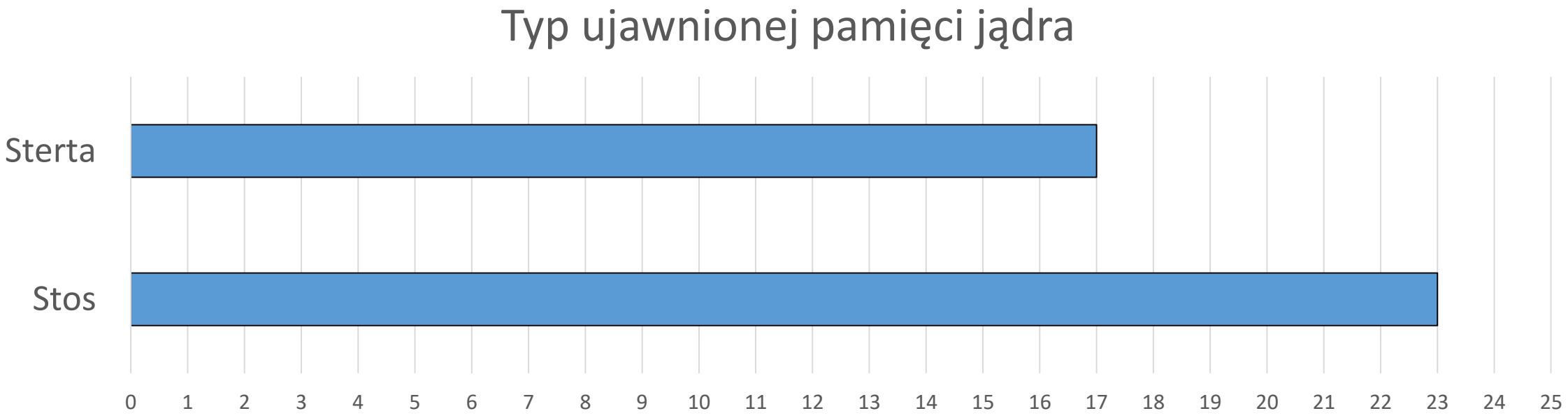
Win32k Information Disclosure Vulnerability	CVE-2017-8677	• fanxiaocao and pjf of IceSword Lab, Qihoo 360 • Mateusz Jurczyk, Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8678	• fanxiaocao and pjf of IceSword Lab, Qihoo 360 • Mateusz Jurczyk, Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8680	Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8681	Mateusz Jurczyk of Google Project Zero
Windows GDI+ Information Disclosure Vulnerability	CVE-2017-8684	Mateusz Jurczyk of Google Project Zero
Windows GDI+ Information Disclosure Vulnerability	CVE-2017-8685	Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8687	Mateusz Jurczyk, Google Project Zero

Windows Information Disclosure Vulnerability	CVE-2017-11817	Mateusz Jurczyk of Google Project zero

Windows Kernel Information Disclosure Vulnerability	CVE-2017-11784	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-11785	Mateusz Jurczyk of Google Project Zero

Podsumowanie odkryć

- Do tej pory **40 podatności** załatanych przez Microsoft od kwietnia do października tego roku.



Zestawienie wycieków ze sterty

CVE	Komponent	Data wydania łatki	Liczba ujawnionych bajtów
CVE-2017-0258	nt!SepInitSystemDacls	Maj 2017	8
CVE-2017-0259	nt!NtTraceControl (EtwpSetProviderTraits)	Maj 2017	60
CVE-2017-8484	win32k!NtGdiGetOutlineTextMetricsInternalW	Czerwiec 2017	5
CVE-2017-8488	Mountmgr, IOCTL_MOUNTMGR_QUERY_POINTS	Czerwiec 2017	14
CVE-2017-8489	WMIDataDevice, IOCTL 0x224000 (WmiQueryAllData)	Czerwiec 2017	72
CVE-2017-8490	win32k!NtGdiEnumFonts	Czerwiec 2017	6672
CVE-2017-8491	Volmgr, IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS	Czerwiec 2017	8
CVE-2017-8492	Partmgr, IOCTL_DISK_GET_DRIVE_GEOMETRY_EX	Czerwiec 2017	4
CVE-2017-8469	Partmgr, IOCTL_DISK_GET_DRIVE_LAYOUT_EX	Czerwiec 2017	484
CVE-2017-8462	nt!NtQueryVolumeInformationFile (FileFsVolumeInformation)	Czerwiec 2017	1
CVE-2017-0299	nt!NtNotifyChangeDirectoryFile	Czerwiec 2017	2
CVE-2017-8564	Nsiproxy, IOCTL 0x120007 (NsiGetParameter)	Lipiec 2017	13
CVE-2017-8668	Volume Manager Extension Driver	Sierpień 2017	?
CVE-2017-8680	win32k!NtGdiGetGlyphOutline	Wrzesień 2017	Dowolna
CVE-2017-11785	nt!NtQueryObject (ObjectNameInformation)	Październik 2017	56
CVE-2017-11784	nt!RtlpCopyLegacyContextX86	Październik 2017	192
CVE-2017-11817	Ntfs!LfsRestartLogFile	Październik 2017	~7600

Zestawienie wycieków ze stosu

CVE	Komponent	Data wydania łatki	Liczba ujawnionych bajtów
CVE-2017-0167	win32kfull!SfnINLPUAHDRAWMENUITEM	Kwiecień 2017	20
CVE-2017-0245	win32k!xxxClientLpkDrawTextEx	Maj 2017	4
CVE-2017-8482	nt!KiDispatchException	Czerwiec 2017	32
CVE-2017-8470	win32k!NtGdiExtGetObjectW	Czerwiec 2017	50
CVE-2017-8471	win32k!NtGdiGetOutlineTextMetricsInternalW	Czerwiec 2017	4
CVE-2017-8472	win32k!NtGdiGetTextMetricsW	Czerwiec 2017	7
CVE-2017-8473	win32k!NtGdiGetRealizationInfo	Czerwiec 2017	8
CVE-2017-8474	DeviceApi (PiDqIrpQueryGetResult, PiDqIrpQueryCreate, PiDqQueryCompletePendedIrp)	Czerwiec 2017	8
CVE-2017-8475	win32k!ClientPrinterThunk	Czerwiec 2017	20
CVE-2017-8485	nt!NtQueryInformationJobObject (BasicLimitInformation, ExtendedLimitInformation)	Czerwiec 2017	8
CVE-2017-8476	nt!NtQueryInformationProcess (ProcessVmCounters)	Czerwiec 2017	4
CVE-2017-8477	win32k!NtGdiMakeFontDir	Czerwiec 2017	104
CVE-2017-8478	nt!NtQueryInformationJobObject (information class 12)	Czerwiec 2017	4
CVE-2017-8479	nt!NtQueryInformationJobObject (information class 28)	Czerwiec 2017	16
CVE-2017-8480	nt!NtQueryInformationTransaction (information class 1)	Czerwiec 2017	6
CVE-2017-8481	nt!NtQueryInformationResourceManager (information class 0)	Czerwiec 2017	2
CVE-2017-0300	nt!NtQueryInformationWorkerFactory (WorkerFactoryBasicInformation)	Czerwiec 2017	5
CVE-2017-8681	win32k!NtGdiGetPhysicalMonitorDescription	Wrzesień 2017	128
CVE-2017-8684	win32k!NtGdiGetFontResourceInfoInternalW	Wrzesień 2017	88
CVE-2017-8685	win32k!NtGdiEngCreatePalette	Wrzesień 2017	1024
CVE-2017-8687	win32k!NtGdiDoBanding	Wrzesień 2017	8
CVE-2017-8677	win32k!NtGdiHLSurfGetInformation (information class 3)	Wrzesień 2017	8
CVE-2017-8678	win32k!NtQueryCompositionSurfaceBinding	Wrzesień 2017	4

Reprodukowanie wycieków ze sterty

- Używamy zwykłej maszyny wirtualnej z Windowsem w odpowiedniej wersji.
- Sprawdzamy, który sterownik alokuje ujawnianą pamięć (np. win32k.sys).
- Włączamy mechanizm **Special Pools** dla tego modułu i restartujemy system.
- Dwukrotnie uruchamiamy PoC i widzimy zmienny, powtarzający się bajt znacznika w miejscu gdzie wyciekają dane.

```
D:\>VolumeDiskExtents.exe
00000000: 01 00 00 00 39 39 39 39 ....9999
00000008: 00 00 00 00 39 39 39 39 ....9999
00000010: 00 00 50 06 00 00 00 00 ..P.....
00000018: 00 00 a0 f9 09 00 00 00 ........
```

```
D:\>VolumeDiskExtents.exe
00000000: 01 00 00 00 2f 2f 2f 2f ....////
00000008: 00 00 00 00 2f 2f 2f 2f ....////
00000010: 00 00 50 06 00 00 00 00 ..P.....
00000018: 00 00 a0 f9 09 00 00 00 ........
```

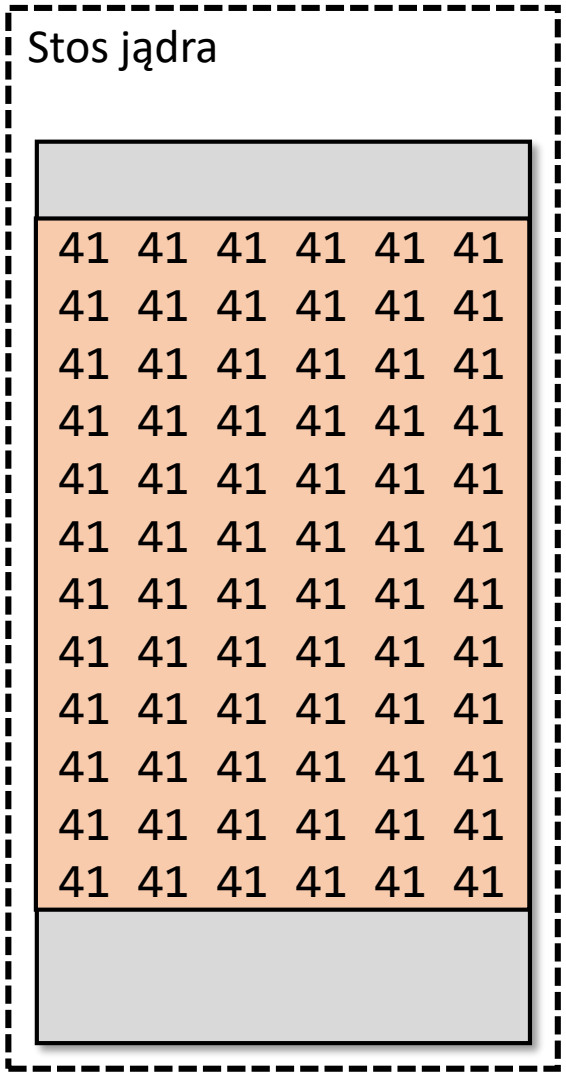
Reprodukowanie wycieków ze stosu

- Znacznie trudniejsze – nie ma oficjalnej/udokumentowanej metody wypełniania stosu kontrolowanym “znacznikiem”.
- W normalnych warunkach trudno określić, które bajty w wyjściu *syscalla* są niezainicjalizowane.
  - Na stosie leżą w większości stałe, mało interesujące dane (np. zera).
  - Obserwacje Microsoftu mogłyby się różnić w ich środowisku testowym.
- Niezawodnie działające programy proof-of-concept są bardzo pożądane.
  - Żeby upewnić się, że błąd faktycznie ma miejsce poza środowiskiem Bochs.
  - Żeby osoba zajmująca się zgłoszeniem widziała dokładnie to samo co my.

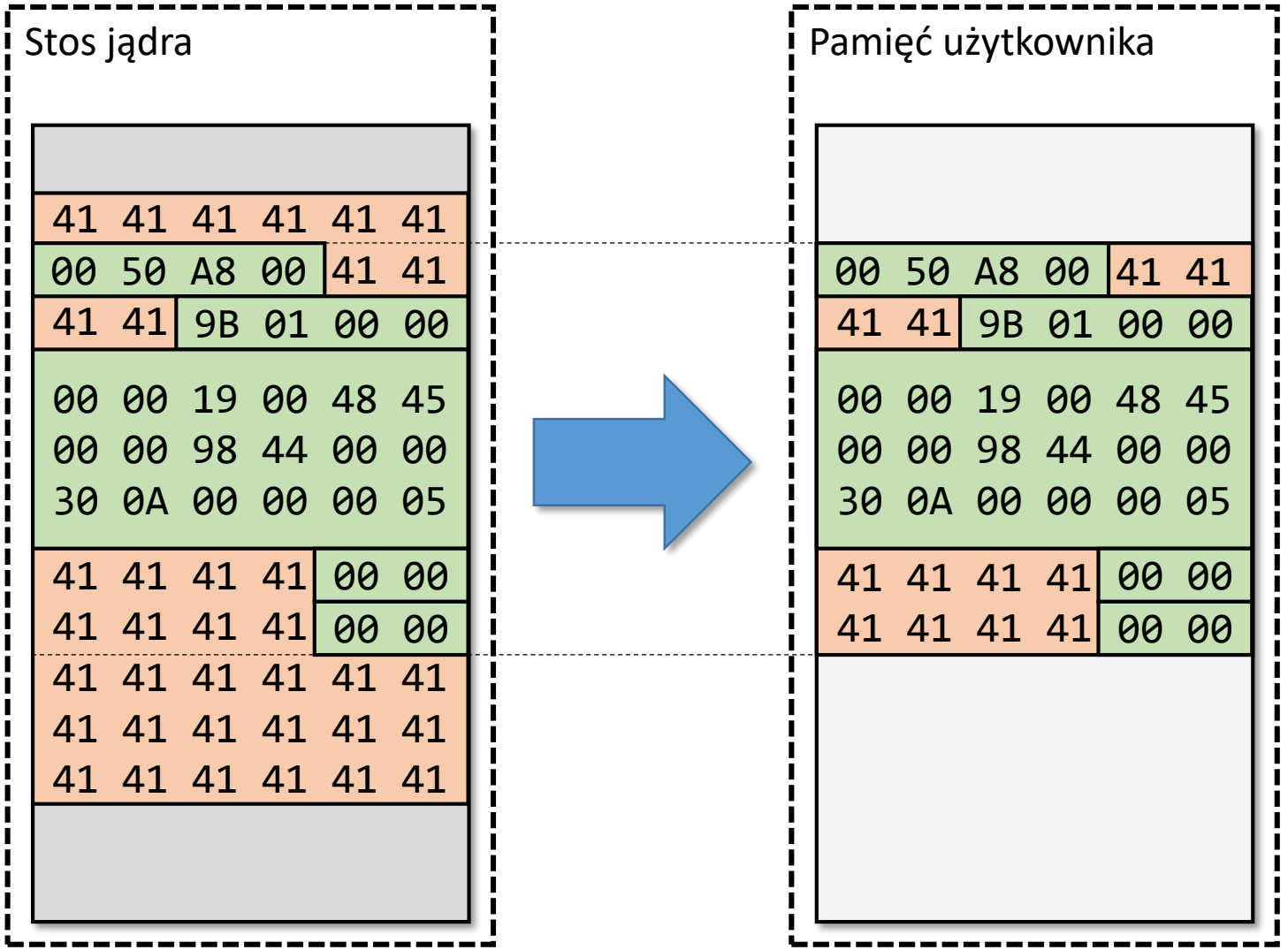
Z pomocą przychodzi *stack spraying*

- W jądrze Windows istnieje kilka funkcji, który kopiują na stos spore regiony danych przekazane przez aplikację.
  - Rodzaj optymalizacji – stosowe bufory używane do „krótkich” zapytań.
- Łatwe do odnalezienia: funkcje o nazwie Nt* z największymi ramkami stosu na liście funkcji w IDA Pro.
- Mój faworyt: **`nt!NtMapUserPhysicalPages`**
  - Wypełnia do 4096 bajtów stosu na x86 i do 8192 bajtów x86-64.
  - Udokumentowane w poście „*nt!NtMapUserPhysicalPages and Kernel Stack-Spraying Techniques*” na blogu w 2011 r.

1. Wypełniamy stos jądra łatwo rozpoznawalnym wzorcem



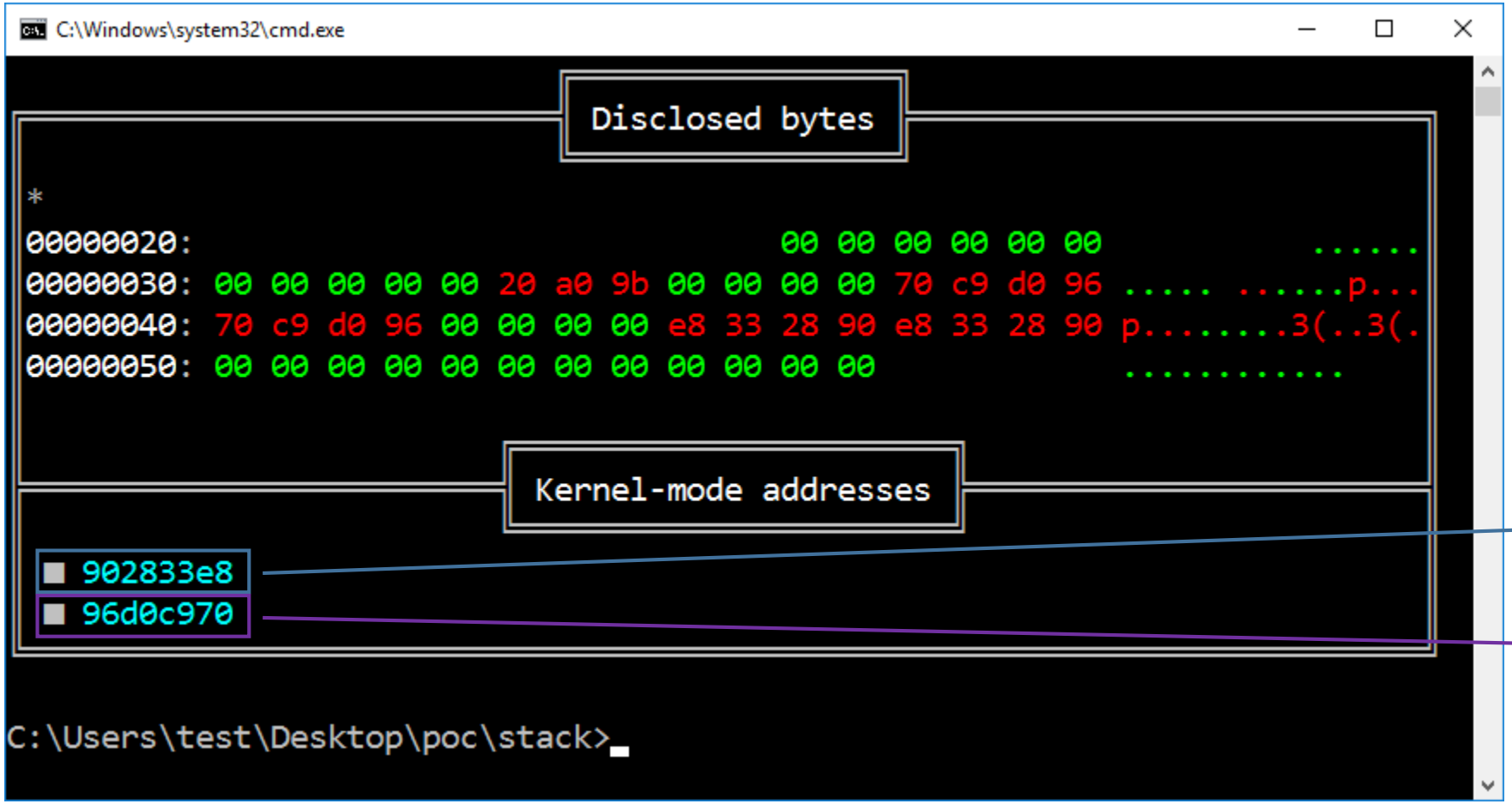
2. Od razu potem wywołujemy wyciek i widzimy bajty naszego wzorca tam, gdzie znajdują się niezainicjalizowane dane



```
D:\>NtGdiGetRealizationInfo.exe
00000000: 10 00 00 00 03 01 00 00 ........
00000008: 2e 00 00 00 69 00 00 46 ....i..F
00000010: 41 41 41 41 41 41 41 41 AAAAAAAA
```

Przykłady

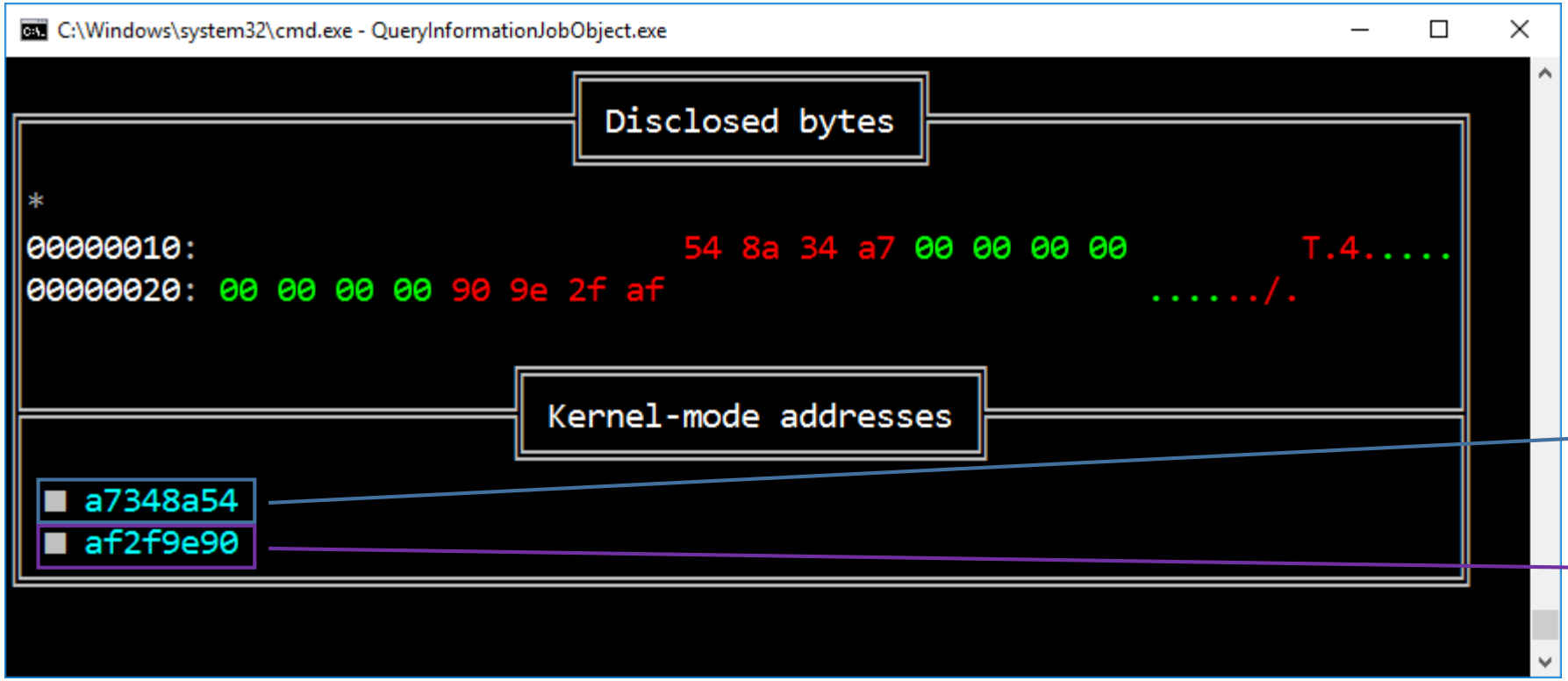
CVE-2017-8470 (win32k!NtGdiExtGetObjectW)



Adres stosu jądra

Adres sterty jądra (*paged pool*)

CVE-2017-8479 (nt!NtQueryInformationJobObject)



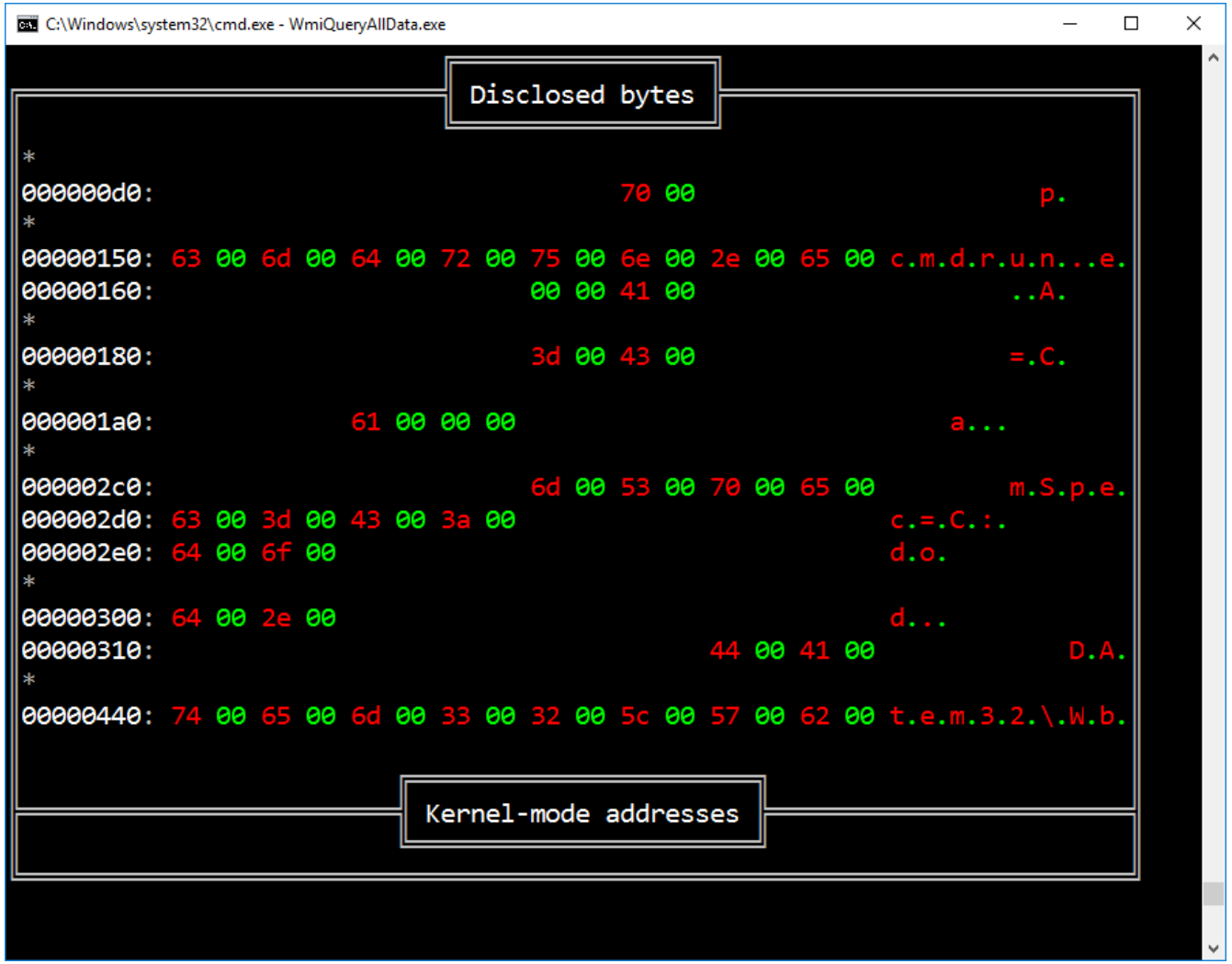
Adres sterownika condrv.sys

Adres sterty jądra (*paged pool*)

CVE-2017-8490 (win32k!NtGdiEnumFonts)

Adres sterty jądra
(*paged session pool*)

CVE-2017-8489 (WmiQueryAllData IOCTL)



Windows – podsumowanie

- Poza kilkoma wyjątkami, problem pozostał praktycznie niezauważony przez wszystkie lata istnienia systemu.
- Windows ma bardzo luźne podejście do kopiowania danych między jądrem a aplikacjami.
- Wierzchołek góry lodowej, błędów tego typu jest prawdopodobnie znacznie więcej.
  - Istnieją setki wywołań `memcpy()` kopiujące dane do przestrzeni user-mode, a każde z nich może potencjalnie prowadzić do wycieku.

W przyszłości

- W samym Bochspwnie jest jeszcze wiele do poprawy:
  - Wsparcie dla architektury x86-64.
    - Nowe błędy związane z większym rozmiarem wskaźników i innych prostych typów.
  - Powiększanie pokrycia kodu jądra – odwieczny problem.
  - Lepsza propagacja *taintu* – możliwe głównie dla Microsoftu.
    - Instrumentacja dodana w czasie kompilacji dużo lepsza od działającej na skompilowanym jądrze.

Bochspwn vs. Linux

Oznaczanie alokacji na stercie

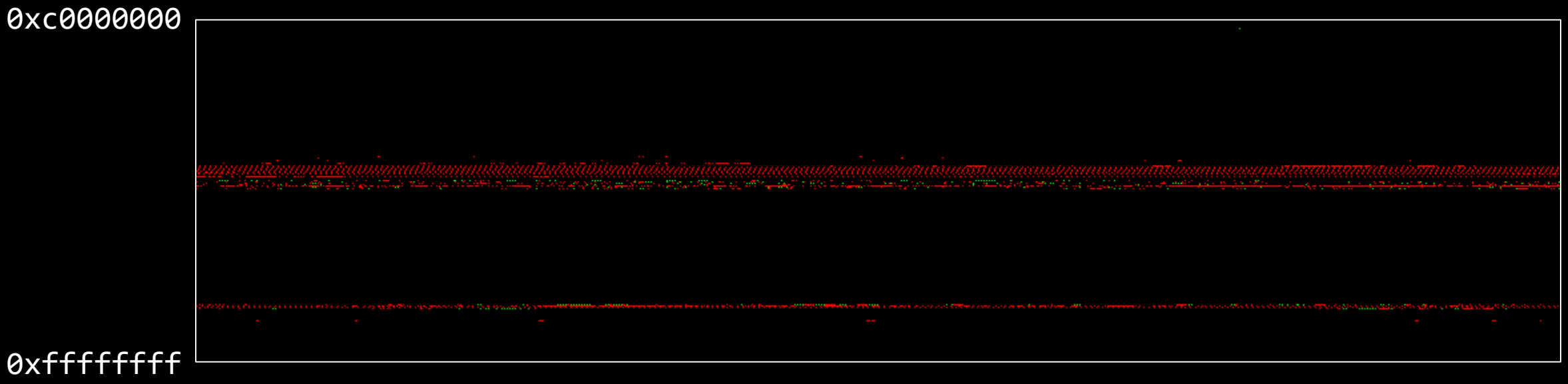
- Znacznie bardziej skomplikowane niż na Windowsie:
  - Wiele różnych alokatorów, publicznych i wewnętrznych, z różnymi wariantami: **`kmalloc`**, **`vmalloc`**, **`kmem_cache_alloc`**.
  - Różne funkcje alokatora mają różne deklaracje.
  - Przekazywanie argumentów poprzez rejestry (**`regparm=3`**) oznacza, że informacje o alokacji nie są dostępne w momencie wykonania instrukcji RET.
  - Obiekty kmem_cache mają zdefiniowany rozmiar alokacji podczas utworzenia, a nie użycia.
  - Obiekty kmem_cache mogą mieć “konstruktory” (oznaczanie regionu odbywa się w innym czasie niż zwracanie go do alokującego kodu).
  - Alokatory mogą zwracać wartości ≤ 0x10 (nie tylko NULL).

Propagacja taintu

- **CONFIG_X86_GENERIC=y** i **CONFIG_X86_USE_3DNOW=n** powodują kompilację `memcpy()` do `rep movs{d,b}`:

```
.text:C13CC43B                  mov     ebx, ecx
.text:C13CC43D                  mov     edi, eax
.text:C13CC43F                  shr     ecx, 2
.text:C13CC442                  mov     esi, edx
.text:C13CC444                  rep movsd
.text:C13CC446                  mov     ecx, ebx
.text:C13CC448                  and     ecx, 3
.text:C13CC44B                  jz      short loc_C13CC44F
.text:C13CC44D                  rep movsb
.text:C13CC44F
.text:C13CC44F loc_C13CC44F:                            ; CODE XREF: memcpy+1B↑j
.text:C13CC44F                  pop     ebx
.text:C13CC450                  pop     esi
.text:C13CC451                  pop     edi
.text:C13CC452                  pop     ebp
.text:C13CC453                  retn
.text:C13CC453 memcpy           endp
```

Układ pamięci Ubuntu 16.04



60 minut działania, interwał 20s., uruchomienie + fuzzer trinity + linux test project

Wykrywanie błędów – copy_to_user

- Flaga **CONFIG_X86_INTEL_USERCOPY=n** powoduje, że `copy_to_user()` jest skompilowane do `rep movs{d,b}`.

```
.text:C13CCA2B                  mov     ebx, ecx
.text:C13CCA2D                  mov     edi, eax
.text:C13CCA2F                  mov     esi, edx
.text:C13CCA31                  cmp     ecx, 7
.text:C13CCA34                  jbe     short loc_C13CCA4E
.text:C13CCA36                  mov     ecx, edi
.text:C13CCA38                  neg     ecx
.text:C13CCA3A                  and     ecx, 7
.text:C13CCA3D                  sub     ebx, ecx
.text:C13CCA3F                  rep movsb
.text:C13CCA41                  mov     ecx, ebx
.text:C13CCA43                  shr     ecx, 2
.text:C13CCA46                  and     ebx, 3
.text:C13CCA49                  nop
.text:C13CCA4A                  rep movsd
.text:C13CCA4C                  mov     ecx, ebx
.text:C13CCA4E
.text:C13CCA4E loc_C13CCA4E:                            ; CODE XREF: __copy_from_user_ll_nocache_nozero+14↑j
.text:C13CCA4E                  rep movsb
.text:C13CCA50                  pop     ebx
.text:C13CCA51                  mov     eax, ecx
.text:C13CCA53                  pop     esi
.text:C13CCA54                  pop     edi
.text:C13CCA55                  pop     ebp
.text:C13CCA56                  retn
.text:C13CCA56 __copy_from_user_ll_nocache_nozero endp
```

Wykrywanie błędów – put_user

- Makro służące do zwracania zmiennych o prostych typach do przestrzeni użytkownika.
- Nie opiera się na `memcpy()`, więc nie działa normalna detekcja.
- Każda architektura – w tym x86 – ma własną implementację.
- Trudno skonwertować makro na użycie `memcpy()`.
  - W argumencie przekazywane są różne konstrukcje: stałe, zmienne, pola struktur, wartości zwracane przez funkcje itd.

Rozwiązanie – tymczasowy *tryb ścisły*

```
#define __put_user(x, ptr) \
({                                          \
        __typeof__(*(ptr)) __x;             \
...
        __asm("prefetcht1 (%eax)");         \
        __x = (x);                          \
        __asm("prefetcht2 (%eax)");         \
...
```

1. Włącz *tryb ścisły* (dla aktualnego ESP)

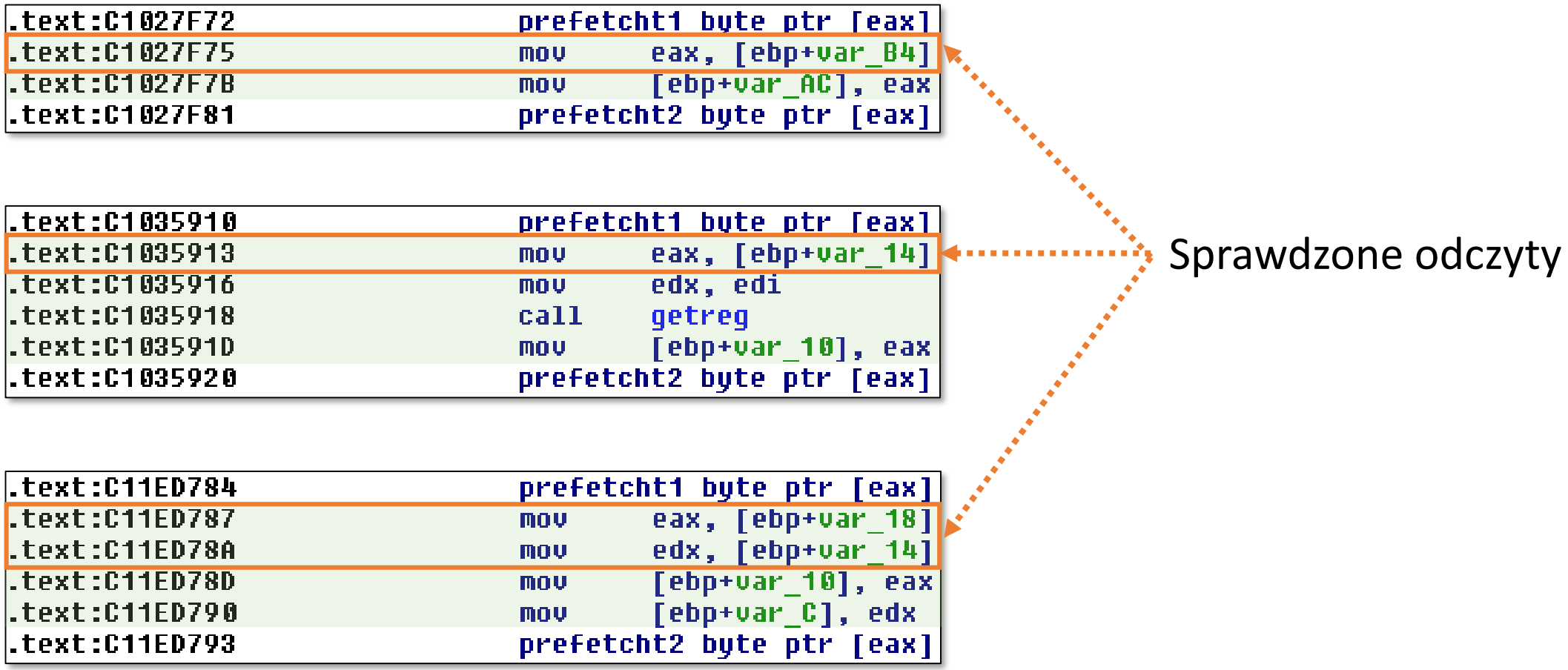
2. Rozwiąż wyrażenie przekazywane klientowi

3. Wyłącz *tryb ścisły*

Tryb ścisły

- Instrukcje **`PREFETCH{1,2}`** działają w Bochs jak NOP.
  - Można wykorzystać je jako znaczniki w kodzie.
- Pomiędzy **PREFETCH1** a **PREFETCH2** wszystkie odczyty niezainicjalizowanej pamięci są zgłaszane jako wycieki kernel→user.
- **365** bloków dodanych do obrazu vmlinux używanego przez Bochspwn.

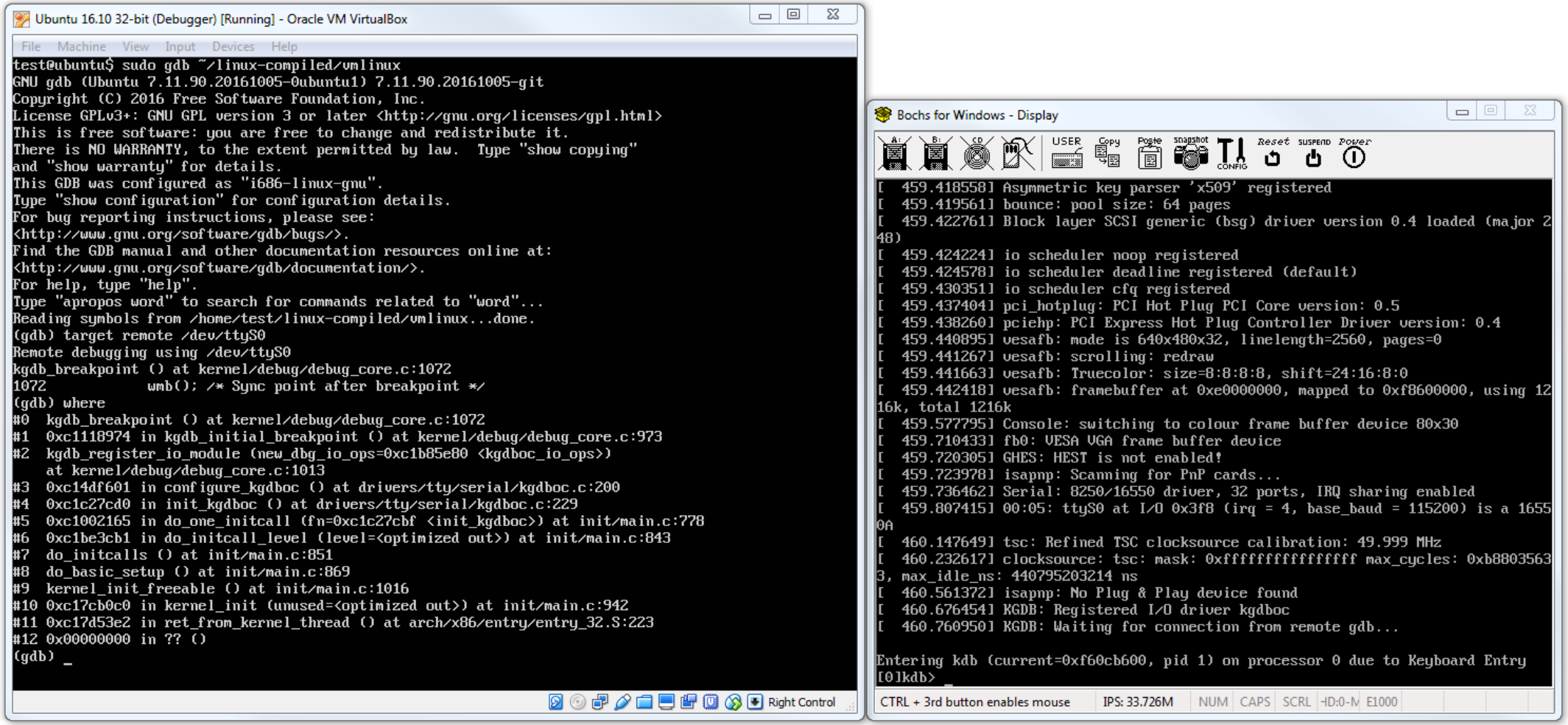
Rozwinięcie put_user widziane w IDA Pro



Przykładowy raport błędu

```
------------------------------ found uninit-access of address f5733f38
========== READ of f5733f38 (4 bytes, kernel--->kernel), pc = f8aaf5c5
                              [          mov edi, dword ptr ds:[ebx+84] ]
[Heap allocation not recognized]
Allocation origin: 0xc16b40bc: SYSC_connect at net/socket.c:1524
Shadow bytes: ff ff ff ff Guest bytes: bb bb bb bb
Stack trace:
#0  0xf8aaf5c5: llcp_sock_connect at net/nfc/llcp_sock.c:668
#1  0xc16b4141: SYSC_connect at net/socket.c:1536
#2  0xc16b4b26: SyS_connect at net/socket.c:1517
#3  0xc100375d: do_syscall_32_irqs_on at arch/x86/entry/common.c:330
  (inlined by) do_fast_syscall_32 at arch/x86/entry/common.c:392
```

Debugowanie jądra



Wykonane testy

- Instrumentacja uruchomiona na **Ubuntu 16.10 32-bit** (**jądro 4.8**).
- Przeprowadzone czynności:
  - Uruchomienie systemu.
  - Logowanie przez SSH.
  - Uruchomienie kilku podstawowych komend i odczytanie pseudo-plików z katalogów **/dev** i **/proc**.
  - Uruchomienie unit testów wchodzących w skład **Linux Test Project** (LTP).
  - Uruchomienie fuzzerów wywołań systemowych **Trinity** i **iknowthis**.
- Najlepiej byłoby użyć fuzzera **Syzkaller** bazującego na informacji o pokryciu kodu, ale brakuje wsparcia dla x86 (tylko x86-64 i arm64).

Wyniki

Bezpośrednie ujawnienia pamięci

- Tylko **jeden** nieznaczny błąd!
- Ujawnienie 7 niezainicjalizowanych bajtów ze stosu w obsłudze niektórych IOCTLi w funkcji **`ctl_ioctl`** (`drivers/md/dm-ioctl.c`).
- Urządzenie **`/dev/control/mapper`**, dostępne wyłącznie dla roota. ☹
- Problem odkryty ok. 20 kwietnia, który miałem zgłaszać kilka dni później, ale…

author Adrian Salido <salidoa@google.com> 2017-04-27 10:32:55 -0700
committer Mike Snitzer <snitzer@redhat.com> 2017-04-27 13:55:13 -0400
commit 4617f564c06117c7d1b611be49521a4430042287 (patch)
tree f8005a09d0eb6827fd541e1c15d3fca1ff85c065
parent 84ff1bcc2e25f1ddf5b350c4fa718ca01fdd88e9 (diff)
download linux-4617f564c06117c7d1b611be49521a4430042287.tar.gz

dm ioctl: prevent stack leak in dm ioctl call

When calling a dm ioctl that doesn't process any data
(IOCTL_FLAGS_NO_PARAMS), the contents of the data field in struct
dm_ioctl are left initialized. Current code is incorrectly extending
the size of data copied back to user, causing the contents of kernel
stack to be leaked to user. Fix by only copying contents before data
and allow the functions processing the ioctl to override.

Cc: stable@vger.kernel.org
Signed-off-by: Adrian Salido <salidoa@google.com>
Reviewed-by: Alasdair G Kergon <agk@redhat.com>
Signed-off-by: Mike Snitzer <snitzer@redhat.com>

Diffstat

-rw-r--r-- drivers/md/dm-ioctl.c 2

1 files changed, 1 insertions, 1 deletions

```
diff --git a/drivers/md/dm-ioctl.c b/drivers/md/dm-ioctl.c
index 0956b86..ddda810 100644
--- a/drivers/md/dm-ioctl.c
+++ b/drivers/md/dm-ioctl.c
@@ -1840,7 +1840,7 @@ static int ctl_ioctl(uint command, struct dm_ioctl __user *user)
        if (r)
                goto out;

-       param->data_size = sizeof(*param);
+       param->data_size = offsetof(struct dm_ioctl, data);
        r = fn(param, input_param_size);

        if (unlikely(param->flags & DM_BUFFER_FULL_FLAG) &&
```

2017-04-27 10:32:55 -0700

Szerszy zakres detekcji

- Bochspwn może wykrywać wszystkie odwołania do niezainicjalizowanej pamięci, nie tylko wycieki do trybu użytkownika.
  - Z dostępem do kodu źródłowego łatwo zrozumieć i przeanalizować każde zgłoszenie.
- Może tutaj bardziej nam się poszczęści?

Użycie niezainicjalizowanej pamięci

Miejsce	Naprawiony	Patch wysłany	Znalezione przez inną osobę	Typ pamięci
`llcp_sock_connect w net/nfc/llcp_sock.c`	Tak	Tak	Tak (po Bochspwnnie)	Stos
`Funkcje obsługi bind() i connect() w różnych rodzajach gniazd (bluetooth, caif, iucv, nfc, unix)`	Tak	Tak	Nie	Stos
`deprecated_sysctl_warning w kernel/sysctl_binary.c`	Tak	Tak	Tak (po Bochspwnie)	Stos
`SYSC_epoll_ctl w fs/eventpoll.c`	Tak	n/a	Tak	Stos
`devkmsg_read w kernel/printk/printk.c`	Tak, w jądrach 4.10+	n/a	Reorganizacja kodu	Sterta
`dnrmg_receive_user_skb w net/decnet/netfilter/dn_rtmsg.c`	Tak	Tak	Nie	Sterta
`nfnetlink_rcv w net/netfilter/nfnetlink.c`	Tak	Tak	Nie	Sterta
`ext4_update_bh_state w fs/ext4/inode.c`	Tak	n/a	Tak	Stos
`nl_fib_lookup w net/ipv4/fib_frontend.c`	Tak	n/a	Tak	Sterta
`fuse_release_common w fs/fuse/file.c`	Tak	Tak	Nie	Sterta
`apply_alternatives w arch/x86/kernel/alternative.c`	Tak	Tak	Nie	Stos
`__bpf_prog_run w kernel/bpf/core.c`	n/a	n/a	Tak	Stos
`crng_reseed w drivers/char/random.c`	n/a	n/a	Nie	Stos
`unmapped_area_topdown w mm/mmap.c`	n/a	n/a	Nie	Stos

Bonus: Lokalny DoS (NULL Pointer Dereference) znaleziony podczas eksperymentów z innym błędem.

Podsumowanie wyników

- Lista jest długa, ale błędy w większości błahe.
  - Na przykład pozwalają na odpowiedzenie na pytanie „czy niezainicjalizowany bajt na stosie jądra jest równy 0?”
  - Jeden typowy błąd ujawnienia pamięci w gniazdach **AF_NFC**.
- Potwierdzenie tego, że podejście działa, po prostu nie ma zbyt wielu podatności do znalezienia.

KernelMemorySanitizer

- Rozwój Linuksa jest bardzo szybki, błędy są naprawiane każdego dnia.
- Większość kolizji nastąpiła z projektem **KMSAN**.
  - Tworzony przez Alexandra Potapenko.
  - Instrumentacja dodawana podczas kompilacji, wykrywająca użycie niezainicjalizowanej pamięci.
  - Bliźniaczy projekt KernelAddressSanitizer, MemorySanitizer (dla user-mode) i wszystkich innych Sanitizerów.
- Poprawne długoterminowe podejście do problemu.

Inne metody

Alternatywne podejścia

1. Ręczny audyt odwołań do `memcpy()` w funkcjach obsługujących wywołania systemowe.
2. Porównywanie wyjść dwukrotnie wywołanego *syscalla* w poszukiwaniu różnic.
3. Bindiffing jądra i sterowników z różnych wersji Windowsa.

CVE-2017-8680
CVE-2017-8681

CVE-2017-8478
CVE-2017-8479
CVE-2017-8480
CVE-2017-8481
CVE-2017-0300

CVE-2017-8684
CVE-2017-8685

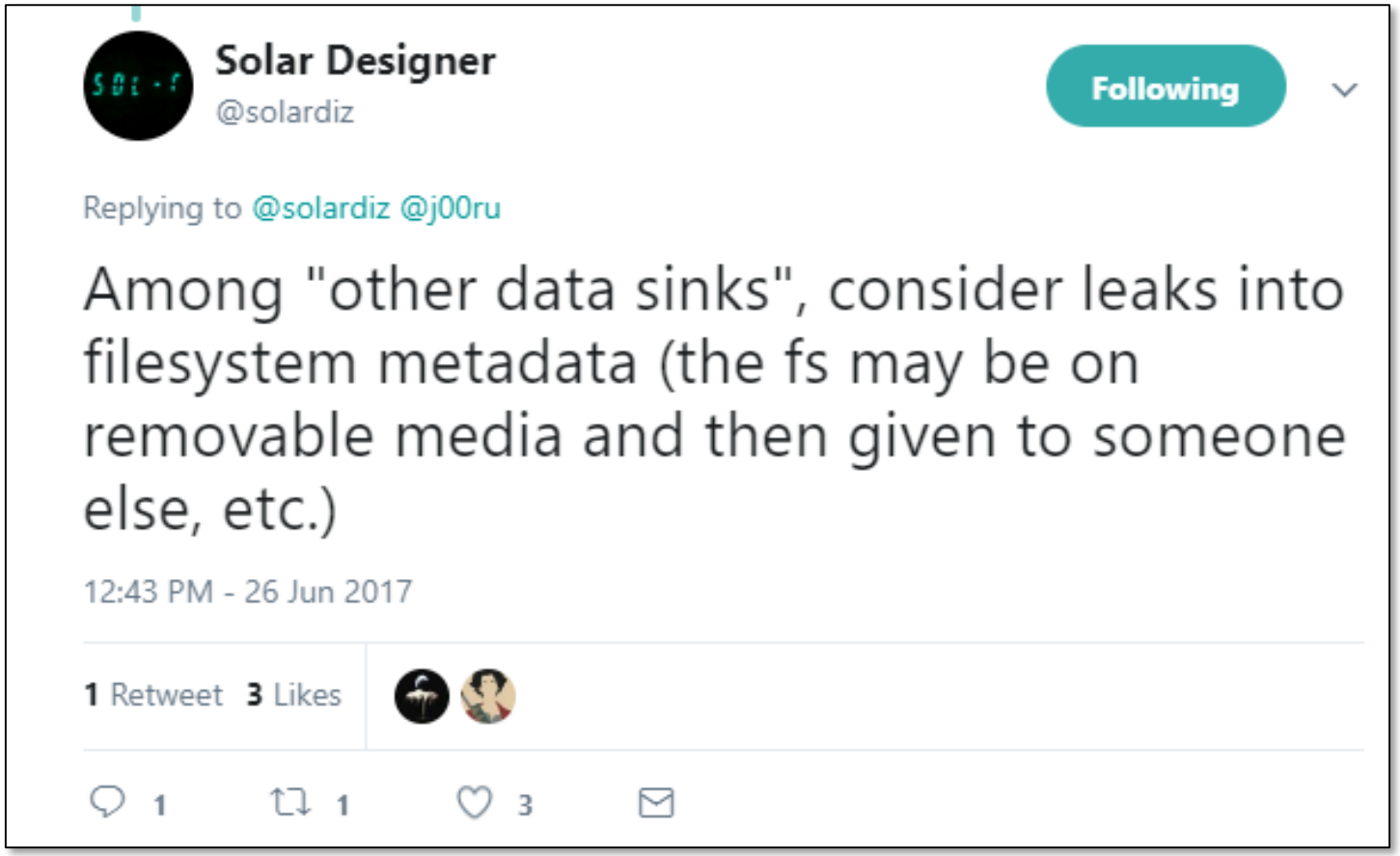
Bochspwn bez *taint trackingu*

- A gdyby nie przechowywać żadnych metadanych i polegać tylko na pamięci gościa?
- Schemat:
  - Wszystkie alokacje na stosie i stercie wypełniamy znanym wzorcem.
  - Przy każdym zapisie kernel→user sprawdzamy, czy zapisywane dane zawierają część wzorca.
- Pomysł wykorzystany w badaniach *Automatically Discovering Windows Kernel Information Leak Vulnerabilities*.

Bochspwn bez *taint trackingu*

- Zalety:
  - Wydajne, dużo mniejszy narzut obliczeniowy i pamięciowy.
  - Możliwość implementacji bez wykorzystania pełnego emulatora x86.
  - Możliwość wykrywania wycieków do innych miejsc – ruchu sieciowego, systemu plików itp.
- Wady:
  - Występowanie *false-positives*, jeśli przypadkowe dane pokryją się ze wzorcem.
  - Trudność w wykrywaniu niewielkich, kilkubajtowych wycieków.

Inne miejsca wycieków



Wycieki do systemu plików

- Systemy plików są reprezentowane przez dość złożone struktury danych.
- Czy możliwe jest, by niektóre sterowniki zapisywały je na nośnik bezpośrednio ze stosu/sterty?
- Fizyczny scenariusz ataku:
  - Atakujący prosi ofiarę o udostępnienie plików na pendrivie lub karcie pamięci.
  - Ofiara kopiuje pliki i przekazuje nośnik atakującemu zakładając, że znajdują się na nim wyłącznie skopiowane dane.
  - Atakujący zrzuca obraz pamięci masowej, zyskując dostęp do fragmentów niezainicjalizowanej pamięci jądra systemu ofiary.

Wykrywanie wycieków na dysk

- Wypełniamy alokacje na stosie i stercie rozpoznawalnym znacznikiem w instrumentacji.
- Zmieniamy tryb działania dysku w pliku bochsrc z „flat” na „volatile”.
- Uruchamiamy system i wykonujemy operacje na dysku twardym.
- Cyklicznie skanujemy changelog w poszukiwaniu znacznika alokacji i analizujemy w którym miejscu systemu plików się znalazł.

Rezultaty

FAT: brak

Rezultaty

FAT32: brak

Rezultaty

exFAT: brak

NTFS

Windows Kernel pool memory disclosure into NTFS metadata ($LogFile) in Ntfs!LfsRestartLogFile

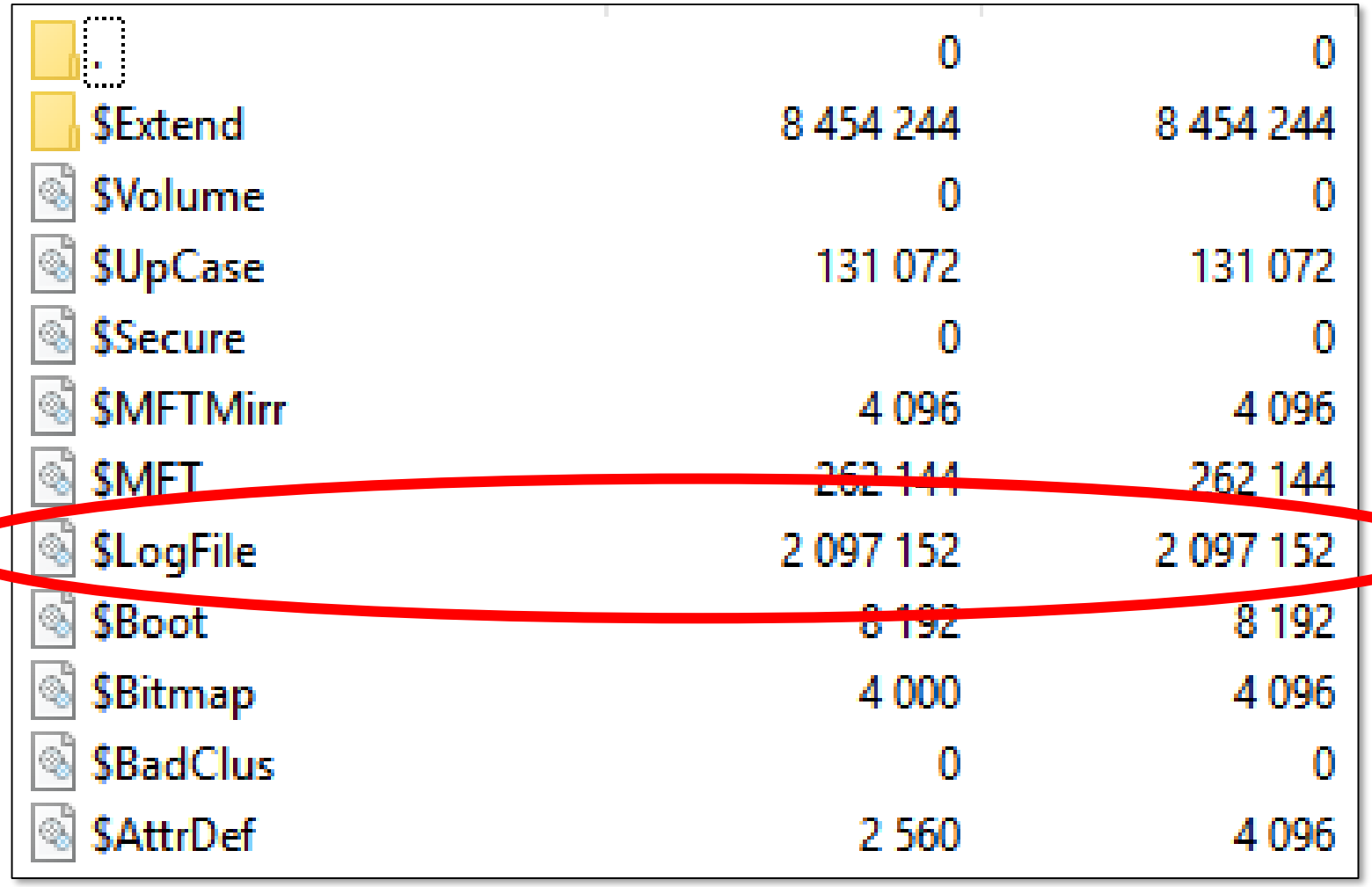
Project Member Reported by mjurczyk@google.com, Sep 4

This tracker entry is a fork of issue #1325, which this bug was reported as a part of. However, as some essential information and context was provided in issue #1325, the "Reported" date was adjusted there to account for it. The new information did not concern the vulnerability discussed here, so we are sticking to the original deadline in this case. Hence the need to create a separate entry in the tracker.

We have discovered that the NTFS.sys driver writes uninitialized kernel pool memory into the internal structures of the file system, while mounting and operating on it. This may be of a security concern in cases where, for example, users share some files with each other via USB sticks or other storage media with NTFS-formatted volumes on them. While the victim would assume that they're only sharing intended data explicitly copied to the disk, they could also unknowingly share bits and pieces of sensitive/confidential information stored in the kernel, that just happened to reside in the memory area used by NTFS.sys while constructing internal file system structures.

Even more scary are leaks which don't require any human interaction and take place immediately when the volume is mounted. In this scenario, it could be possible to disclose kernel memory of a locked machine with physical access to a USB port, by repeatedly plugging in a flash drive (or a device which emulates one), waiting for the uninitialized memory to be written by the system, reading it back and re-mounting the disk.

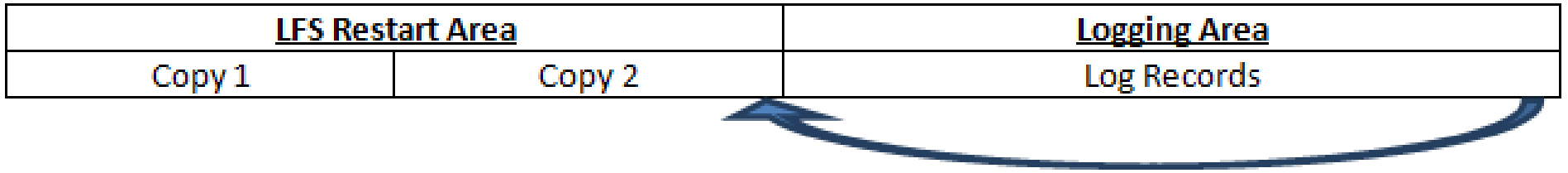
We have implemented some dedicated logic in our Bochspwn system instrumentation to detect instances of such info leaks. As a result, we have found that pool memory allocated in the Ntfs!LfsRestartLogFile function is leaked to NTFS volumes (both system and external ones). From a security perspective, the leak is quite severe, as it is triggered with no user interaction (just by mounting an NTFS volume), and it discloses ~7.5kB of kernel pool memory in 2 chunks of ~3800 consecutive bytes. This data is saved in the first 8192 bytes of the special $LogFile file, as part of the two "RSTR" restart area records. As we understand it, this means that every NTFS-formatted USB stick last accessed by Windows 7 now includes more than 7kB of junk kernel memory from that computer. However, since $LogFile is an internal file, reading from it requires raw disk access which significantly reduces the impact of the bug.



.	0	0
$Extend	8 454 244	8 454 244
$Volume	0	0
$UpCase	131 072	131 072
$Secure	0	0
$MFTMirr	4 096	4 096
$MFT	262 144	262 144
$LogFile	2 097 152	2 097 152
$Boot	8 192	8 192
$Bitmap	4 000	4 096
$BadClus	0	0
$AttrDef	2 560	4 096

CVE-2017-11817: wyciek w $LogFile

- Każda partycja NTFS zawiera wewnętrzny plik $LogFile.
  - Niedostępny z poziomu aplikacji, używany bezpośrednio przez sterowniki.
  - Możliwy do odczytania przy surowym dostępie do urządzenia.
- Inicjalizowany bezpośrednio w momencie montowania systemu plików.



Źródło: http://www.ntfs.com/transaction.htm

Restart areas

- Oba *restart area* to obszary 4096 bajtów w nagłówku $LogFile.
- Alokowane ze sterty w `Ntfs!LfsRestartLogFile`.
  - Region nie zerowany na Windows 7 i starszych systemach.
  - W większości nie inicjalizowany żadnymi danymi przed zapisaniem.
- Ponad 7 kB „śmieci” z pamięci jądra zapisywane automatycznie za każdym podłączeniem zewnętrznego dysku!
- Każdy pendrive itp. podłączony ostatnio do Windows 7 (przed październikową łatką) zawiera ujawnione dane.

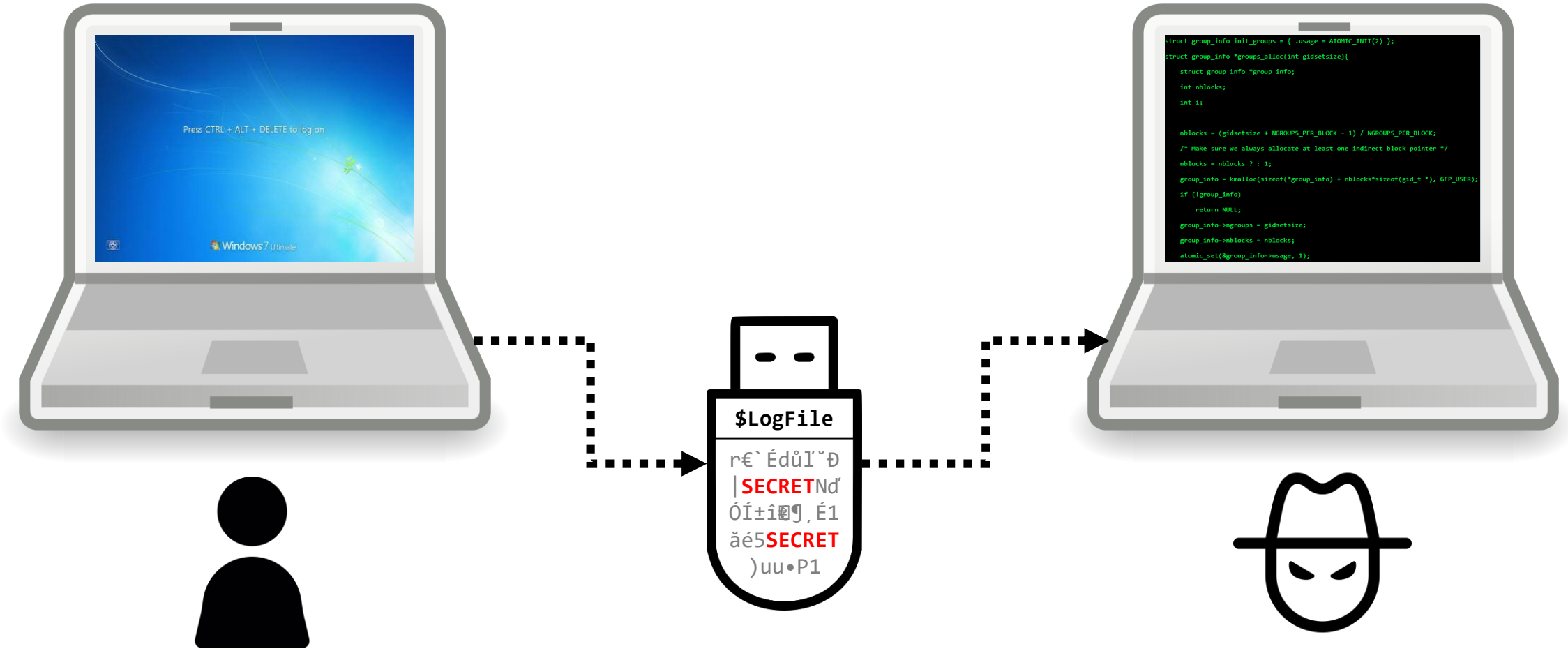
```
PAGE:001B8240 loc_1B8240:                             ; CODE XREF: LfsRestartLogFile(x,x,x,x,x,x,x,x,x,x,x)+882↑j
PAGE:001B8240                 push    'rsfL'          ; Tag
PAGE:001B8245                 imul    eax, [edi+18h], 22h
PAGE:001B8249                 shr     eax, 1
PAGE:001B824B                 push    eax             ; NumberOfBytes
PAGE:001B824C                 push    210h            ; PoolType
PAGE:001B8251                 mov     esi, ds:ExAllocatePoolWithTag(x,x,x)
PAGE:001B8257                 call    esi ; ExAllocatePoolWithTag(x,x,x)
PAGE:001B8259                 mov     [edi+198h], eax
PAGE:001B825F                 push    dword ptr [edi+18h] ; Size
PAGE:001B8262                 push    ebx             ; Val
PAGE:001B8263                 push    eax             ; Dst
PAGE:001B8264                 call    _memset
```

Brakujący `memset()` wprowadzony w Windows 8

Scenariusz bez udziału ofiary

- Windows automatycznie montuje systemy plików fizycznie podłączonych urządzeń.
  - Nawet kiedy komputer jest zablokowany.
- Podatność umożliwia „wysysanie” danych z pamięci jądra przez port USB.
  - Prosto: przełączając pendrive między komputerem ofiary i własnym.
  - Lepiej: przełączając dysk zew. zawierający 20+ partycji NTFS naraz.
  - Najlepiej: używając urządzenia udającego dysk twardy, włączającego i wyłączającego się możliwie szybko w celu maksymalizacji przepustowości wycieku.

Demo


Press CTRL + ALT + DELETE to log on
Windows 7 Ultimate
struct group_info init_groups = { .usage = ATOMIC_INIT(2) };
struct group_info *groups_alloc(int gidsetsize){
struct group_info *group_info;
int nblocks;
int i;
nblocks = (gidsetsize + NGROUPS_PER_BLOCK - 1) / NGROUPS_PER_BLOCK;
/* Make sure we always allocate at least one indirect block pointer */
nblocks = nblocks ? : 1;
group_info = kmalloc(sizeof(*group_info) + nblocks*sizeof(gid_t *), GFP_USER);
if (!group_info)
return NULL;
group_info->ngroups = gidsetsize;
group_info->nblocks = nblocks;
atomic_set(&group_info->usage, 1);
$LogFile
r€`ÉdůĬ˘Đ
|SECRETNď
ÓÍ±îę¶¸É1
ăé5SECRET
)uu•P1

Dziękuję!

@j00ru

http://j00ru.vexillium.org/

j00ru.vx@gmail.com