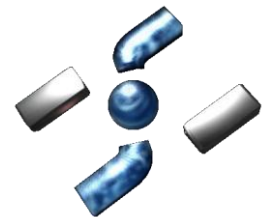


# x86 vs x64 – architektura procesora a exploitacja w systemie Windows

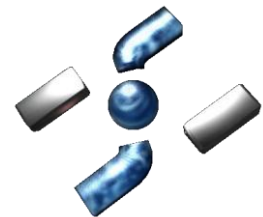
Mateusz „j00ru” Jurczyk  
SecDay, Wrocław 2010





# O mnie

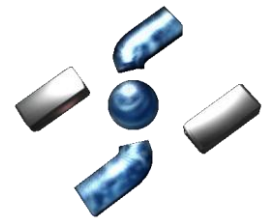
- Reverse engineer @ **Hispacec**
- Vulnerability researcher
- **Vexillum** (<http://vexillum.org>)
- Autor bloga <http://j00ru.vexillum.org/>



# O czym dzisiaj

- x86-64 – z czym mamy do czynienia?
- Podstawowe zmiany architektury
- Exploitacja user-mode
  - Obsługa wyjątków a x64
  - DEP, ASLR
  - ROP
- Różności
- Pytania

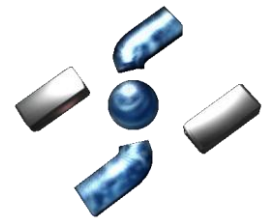
# **SŁÓW KILKA O X86-64**



# x86-64 – o czym mowa

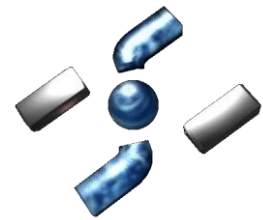
- „Nowa” architektura procesorów 😊
- Rozszerzenie starszych, 32-bitowych procesorów x86
  - Zapewniona wsteczna kompatybilność (tryby *real*, *protected*, *long*)
- Podstawowa jednostka danych: 64 bity / 8 bajtów

# x86-64 – o czym mowa



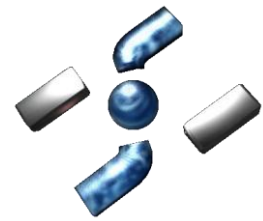
- Od kilku lat produkowane na szeroką skalę
  - Lata 2001 – 2003: pierwsze 64-bitowe procesory Intela i AMD wchodzą na rynek
- Najpopularniejsze systemy operacyjne są portowane na platformę x64
  - W tym Windows – poczynając od Windows XP / 2003 (NT 5.2)

# x86-64 – dobry cel badań?



- „Młoda” architektura, nie zawsze poprawnie rozumiana przez developerów
- Błędy powstałe w toku portowania aplikacji
- Jądro Windows działające wyłącznie w 64-bit
  - Kernel to bardzo dobry cel 😊
- Aplikacje codziennej ~~exploitacji~~ użytku przenoszone na nową platformę

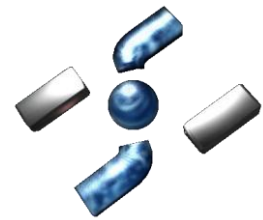
# X86-64 – dobry cel badań?



- x64-specific vulnerabilities
  - **CVE-2007-4573**: Linux kernel IA32 System Call Emulation Vulnerability
  - **CVE-2009-0029**: Linux Kernel 64 Bit ABI System Call Parameter Privilege Escalation Vulnerability
  - **CVE-2010-0010**: Apache 1.3 mod\_proxy HTTP Chunked Encoding Integer Overflow Vulnerability
  - Maaaaaasa innych błędów...

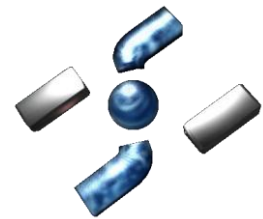


# x86-64 – dobry cel badań?



- Możliwość omijania mechanizmów charakterystycznych dla x64
  - ***Driver Signing Enforcement*** a słynny *pagefile attack* (Joanna Rutkowska, 2006)
  - Zabawy z ***PatchGuard*** (Uninformed 3, 6, 8)

x86-64 – dobry cel badań?



64-bity to przyszłość 😊

# **X86 A X86-64 – ZMIANY ARCHITEKTURALNE**



# Podstawowe zmiany architektury

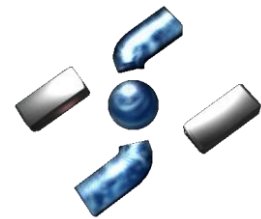
- Rozszerzenie rozmiaru istniejących rejestrów bazowych

EAX (32) → RAX(64)

EFLAGS(32) → RFLAGS(64)

EIP(32) → RIP(64)

RAX			
	EAX		
		AX	
		AH	AL



# Zmiany architektury

- Nowe stare rejestry
  - BPL, SPL, SIL, DIL

RBP			
	EBP		
		BP	
			BPL

RSI			
	ESI		
		SI	
			SIL



# Podstawowe zmiany architektury

- Wprowadzenie ośmiu nowych rejestrów bazowych R8 – R15

R8			
	R8D		
		R8W	
		?	R8B

R15			
	R15D		
		R15W	
		?	R15B



# Podstawowe zmiany architektury

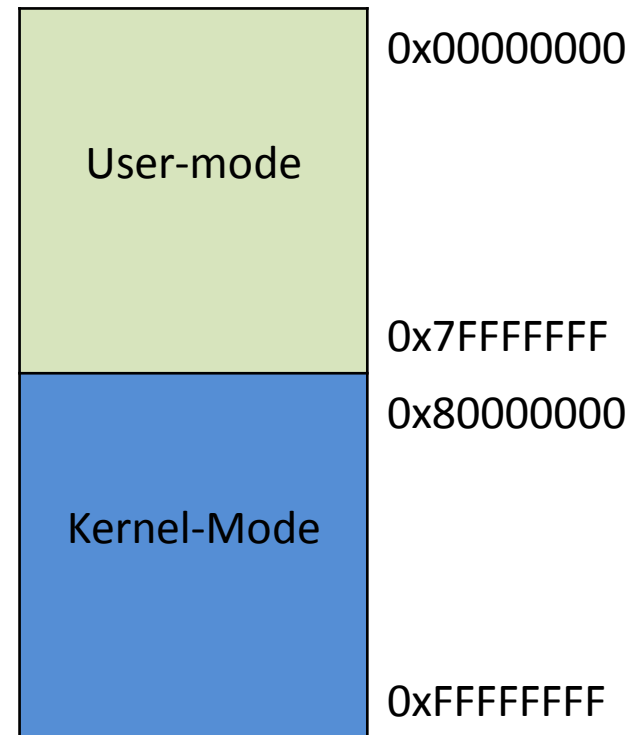
- Adresowanie pamięci wirtualnej
  - Teoretycznie:  $2^{64}$  bajtów adresowalnej pamięci (16 eksabajtów)
  - Praktycznie: wykorzystuje się dolne 48 bitów (256 terabajtów)
    - Pozostała część to *sign-extend* właściwego adresu



# Podstawowe zmiany architektury

Adresowanie pamięci wirtualnej – tryb chroniony (32bit)

- Ciągły model pamięci
- Przestrzeń użytkownika – niskie obszary (2GB)
- Przestrzeń jądra – wysokie obszary (2GB)



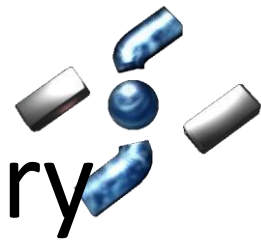




# Podstawowe zmiany architektury

Adresowanie pamięci wirtualnej – *long mode* (64-bit)

- Kanoniczny zapis adresów
  - 6 młodszych bajtów (48 bitów) przeznaczone dla efektywnego adresu
  - Pozostała część **musi** być równa najstarszemu bitowi ( $\text{address}_{47}$ )



# Podstawowe zmiany architektury

## Adresowanie pamięci wirtualnej – *long mode*

1111111111111111	1	00000000000010010000000001101011101100000000000
------------------	---	---

16 bitów dopełnienia

48 bitów - właściwy adres wirtualny

Przykładowy adres trybu jądra

0000000000000000	0	111111110000010000111000010100101100010101010000
------------------	---	--

16 bitów dopełnienia

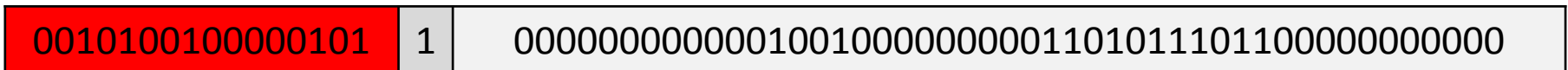
48 bitów - właściwy adres wirtualny

Przykładowy adres trybu użytkownika



# Podstawowe zmiany architektury

Adresowanie pamięci wirtualnej – *long mode*

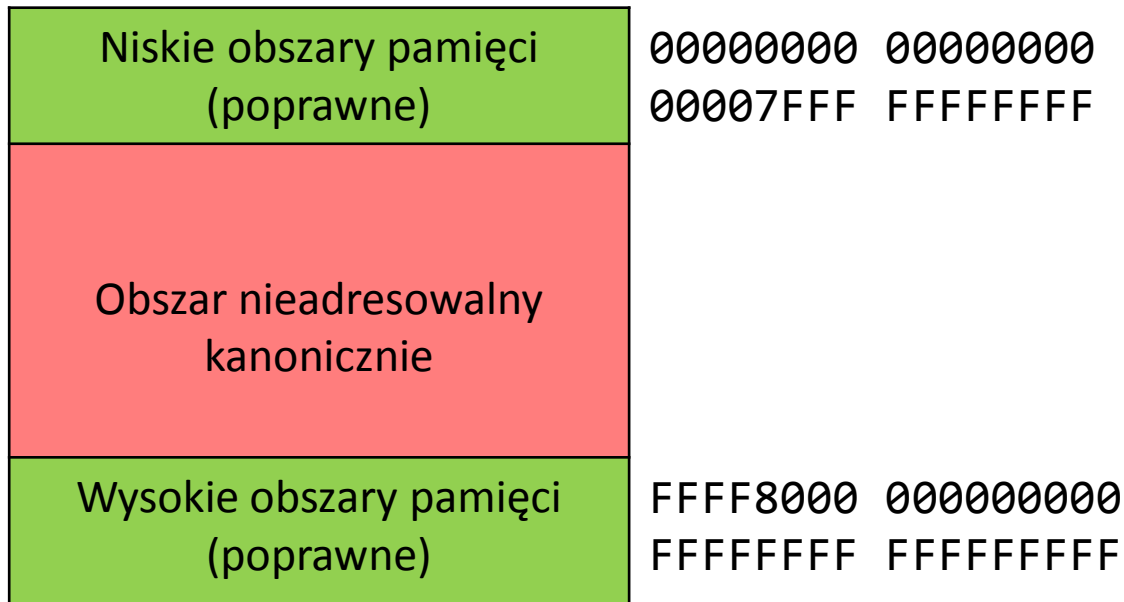


Błędny zapis adresu wirtualnego

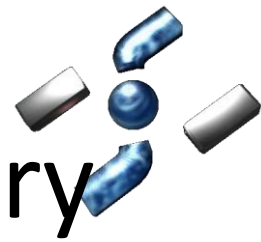


# Podstawowe zmiany architektury

Adresowanie pamięci wirtualnej – *long mode*



Podział przestrzeni pamięci adresowalnej, w trybie 64-bit



# Podstawowe zmiany architektury

- Możliwość odwoływania się do rejestru RIP (wskaźnik instrukcji)
  - Wcześniej wyłącznie pośrednio; głównie przy użyciu instrukcji CALL



# Podstawowe zmiany architektury

Przykład – pobranie adresu aktualnie wykonywanej instrukcji

<b>EAX ← EIP (x86)</b>	<b>RAX ← RIP (x86-64)</b>
<code>call @f</code>	<code>lea rax, [rel 0]</code>
<code>@@:</code>	<code>...</code>
<code>pop eax</code>	<code>...</code>
<code>...</code>	
<code>...</code>	



# Podstawowe zmiany architektury

Przykład – pobranie adresu ciągu tekstowego, znajdującego się w dowolnym miejscu w pamięci

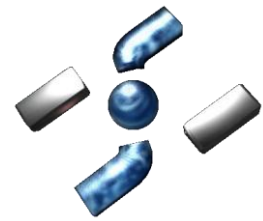
**EAX ← string address (x86)**

```
jmp @1
@2:
pop eax
...
...
@1:
call @2
db 'exemplary text',0
```

**RAX ← string address (x86-64)**

```
lea rax, [rel text_label]
...
...
text_label:
db 'exemplary text',0
```

# Zmiany architektury



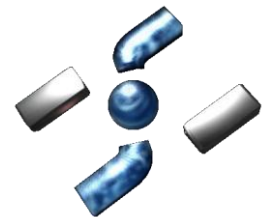
## Rejestry segmentowe

CS, SS, ES, DS  $\rightarrow \emptyset$

FS, GS  $\rightarrow$  TIB, KPCR

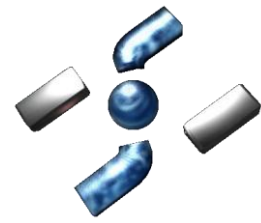


# **USER-MODE EXPLOITATION**



# User-mode exploitation

- Przepięnienia stosu – pamiętamy, co to?



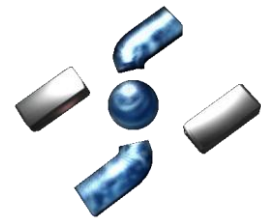
# User-mode exploitation

- Przepętnienia stosowe – pamiętamy, co to?



```
void func(char* str)
{
    int a,b;
    char buf[16];

    try
    {
        ...
    }
}
```



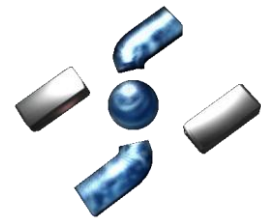
# User-mode exploitation

- Przepętnienia stosowe – pamiętamy, co to?



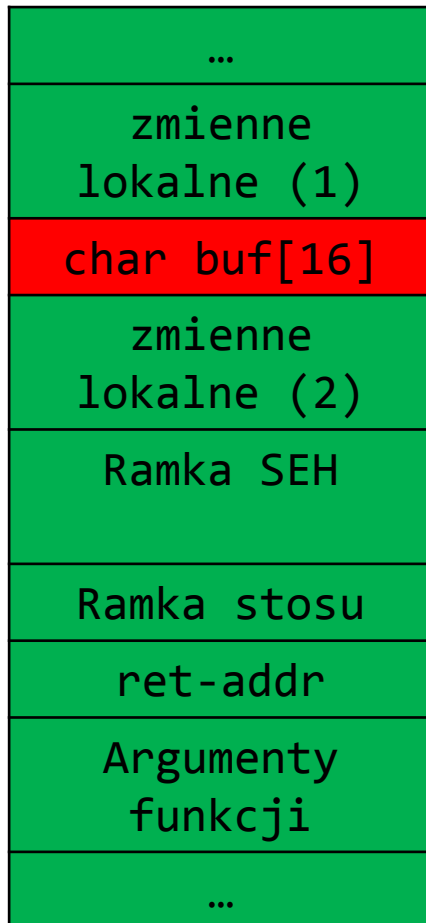
```
void func(char* str)
{
    int a,b;
    char buf[16];

    try
    {
        strcpy(buf, str);
        ...
    }
}
```



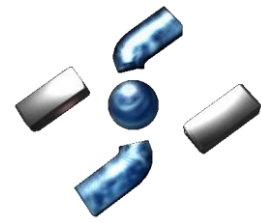
# User-mode exploitation

- Przepięnienia stosowe – pamiętamy, co to?



```
void func(char* str)
{
    int a,b;
    char buf[16];

    try
    {
        strcpy(buf, str);
        ...
    }
}
```



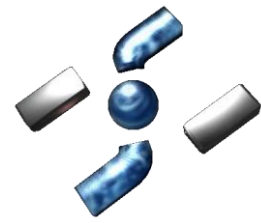
# User-mode exploitation

- Przepętnienia stosowe – pamiętamy, co to?



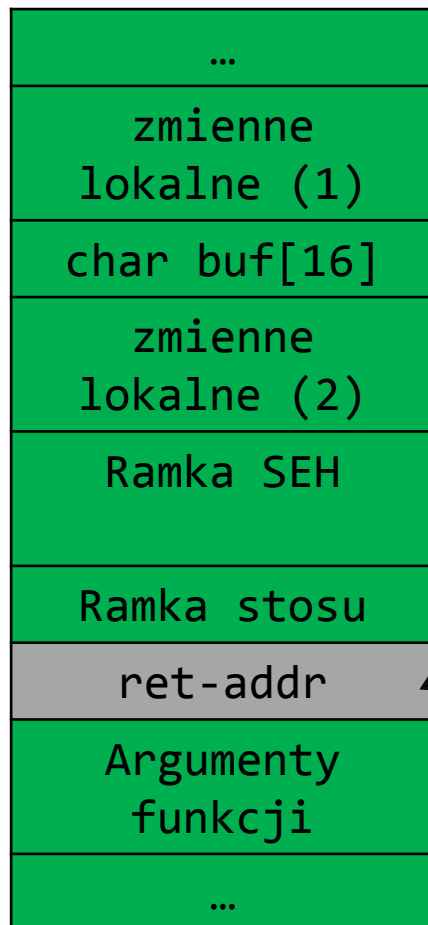
```
void func(char* str)
{
    int a,b;
    char buf[16];

    try
    {
        strcpy(buf, str);
        ...
    }
}
```



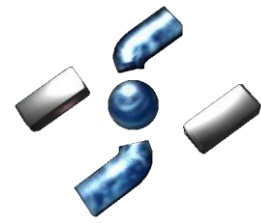
# User-mode exploitation

- Możliwości wykonania payloadu



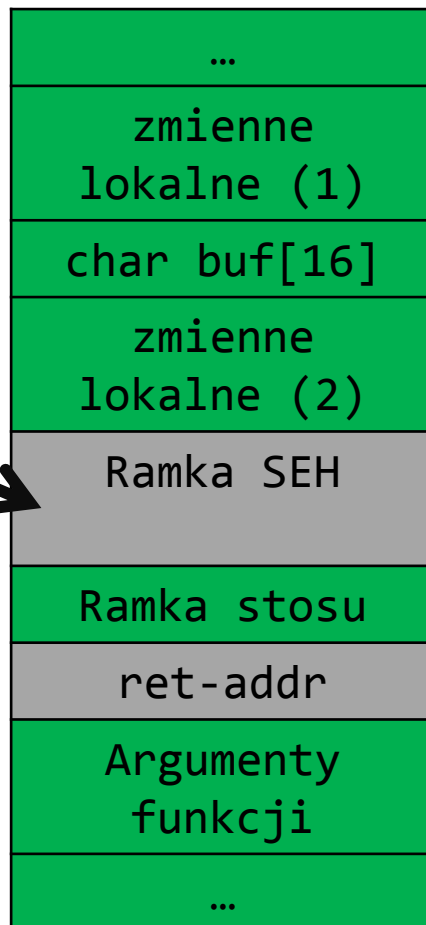
## 1. Adres powrotu funkcji (ret-addr)

Wykonywanie przenoszone do payload'u w momencie wykonania instrukcji **RET**



# User-mode exploitation

- Możliwości wykonania payloadu

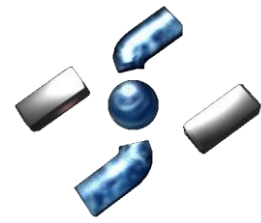


1. Adres powrotu funkcji (ret-addr)

2. Ramka obsługi wyjątków (SEH)

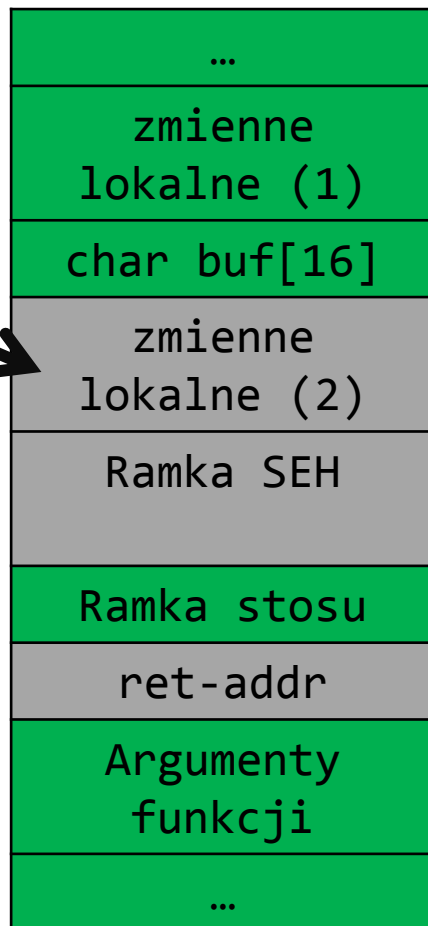
Wykonywanie przenoszone do payload'u w momencie zaistnienia wyjątku w obrębie aktualnego wątku





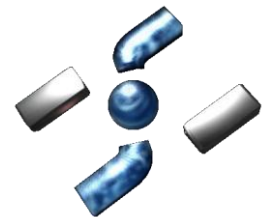
# User-mode exploitation

- Możliwości wykonania payloadu



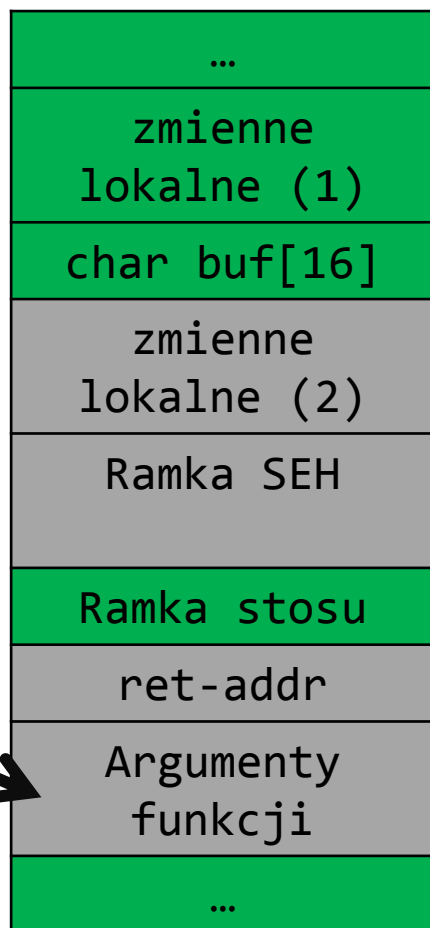
1. Adres powrotu funkcji (ret-addr)
2. Ramka obsługi wyjątków (SEH)
3. Potencjalnie – wskaźniki znajdujące się wśród lokalnych zmiennych

Wykonywanie przenoszone do payload'u w momencie wywołania nadpisanego wskaźnika



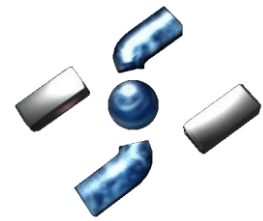
# User-mode exploitation

- Możliwości wykonania payloadu

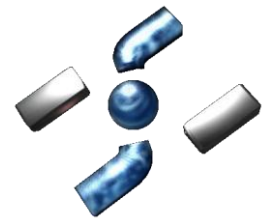


1. Adres powrotu funkcji (ret-addr)
2. Ramka obsługi wyjątków (SEH)
3. Potencjalnie – wskaźniki znajdujące się wśród lokalnych zmiennych
4. Potencjalnie – wskaźniki wśród argumentów funkcji

# User-mode exploitation



# Zabezpieczenia?

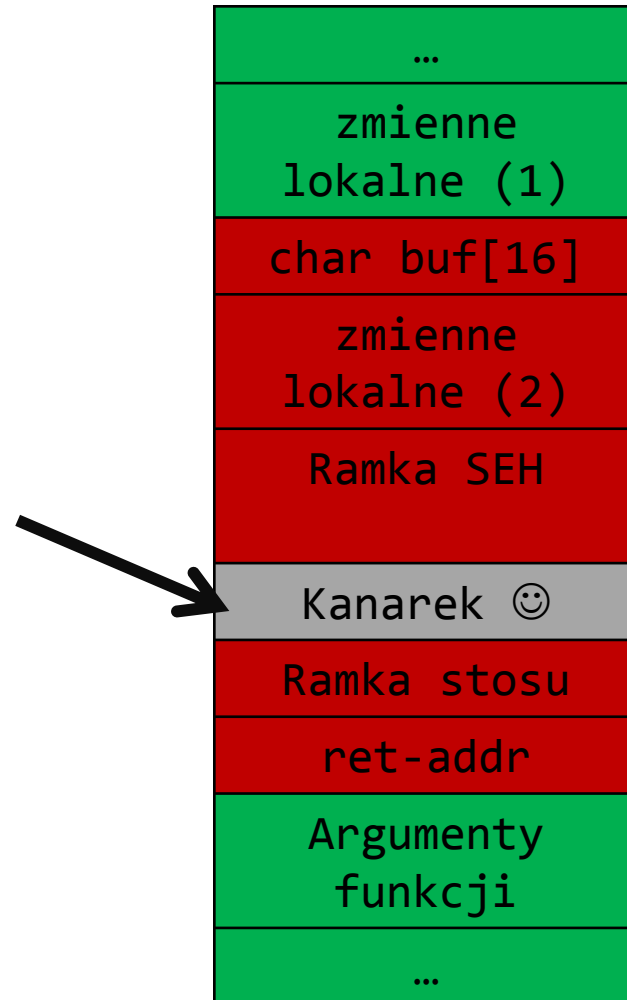
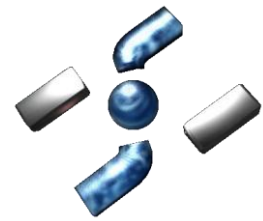


# User-mode exploitation

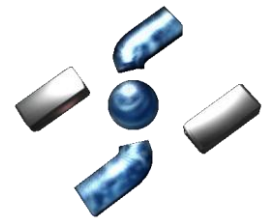
## Zabezpieczenia

- Nadpisanie adresu powrotu
  - Ochrona stosu /GS cookie protection stosowana przez kompilatory Microsoft
  - Inne odmiany kanarków (ProPolice, itd.)

# User-mode exploitation

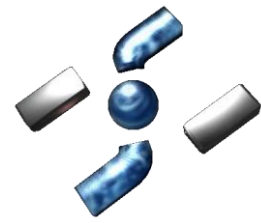


# User-mode exploitation



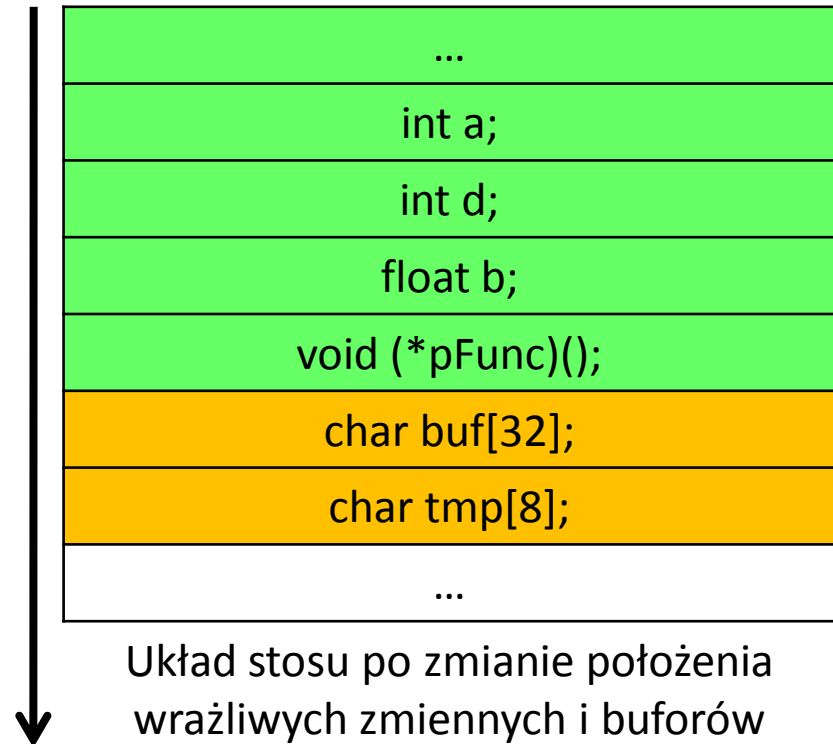
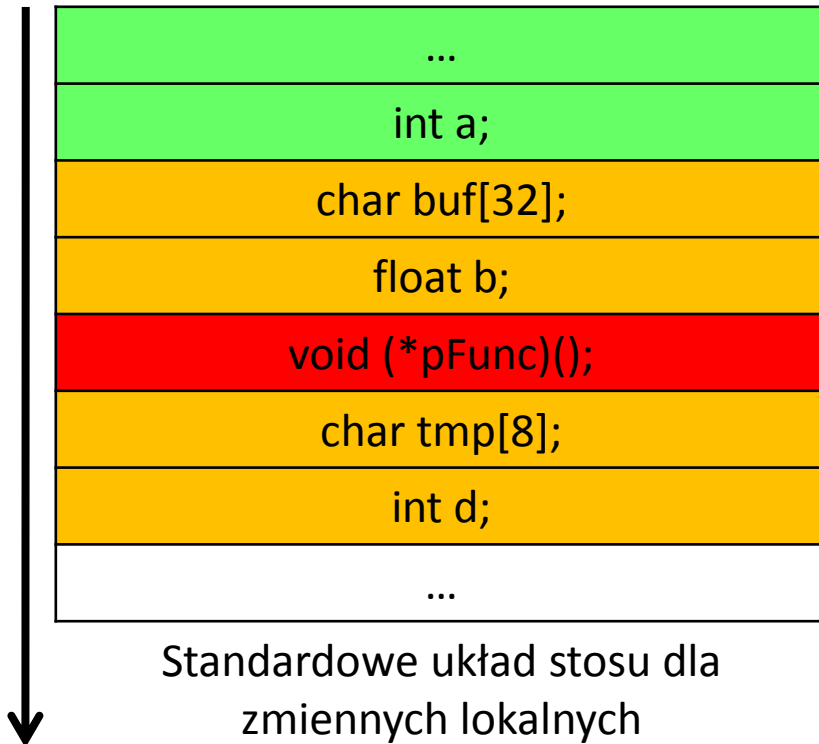
## Zabezpieczenia

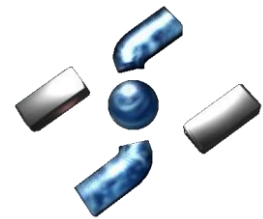
- Nadpisanie zmiennej lokalnej – wskaźnika funkcji
  - *Stack re-ordering* – układ zmiennych, uniemożliwiający nadpisanie wrażliwych wartości (wskaźników)



# User-mode exploitation

## Stack re-ordering - przykład



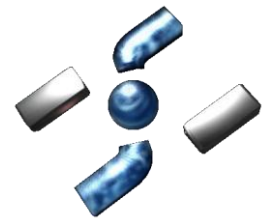


# User-mode exploitation

## Zabezpieczenia

- Nadpisanie argumentów funkcji, będących wskaźnikami
  - Kopiowanie wartości wszystkich argumentów funkcji na stos, *przed* niebezpiecznymi buforami



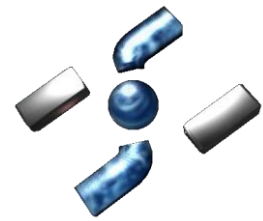


# User-mode exploitation

A co z SEH? 😊

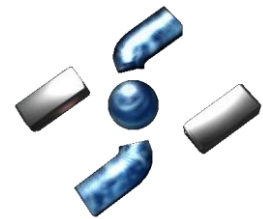
- Również kontrowane przez rozmaite zabezpieczenia
  - SafeSEH
  - SEHOP

# User-mode exploitation



A co z SEH? 😊

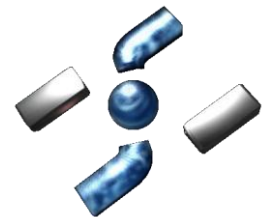
- Exploity wciąż korzystają z obsługi wyjątków
  - SnackAmp 3.1.3B Malicious SMP Buffer Overflow Vulnerability (SEH)
  - MP3 Workstation Version 9.2.1.1.2 SEH exploit (MSF)
  - AoA Audio Extractor Remote ActiveX SEH JIT Spray Exploit (ASLR+DEP Bypass)
  - Wiele innych



# User-mode exploitation

## Obsługa wyjątków a x86-64

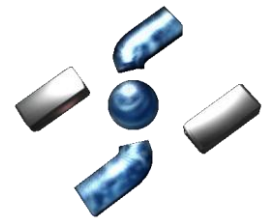
- Brak mechanizmu SEH
- Statyczna lista handlerów
  - Znajduje się w nagłówkach pliku PE32+
  - Rozszerzalne za pomocą API: *RtlAddFunctionTable*, *RtlInstallFunctionTableCallback*



# User-mode exploitation

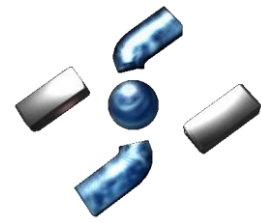
Wnioski?

... trudniej ... 😊



# User-mode exploitation

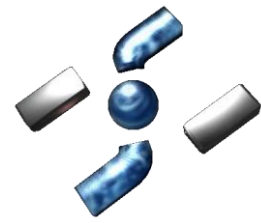
- ALSR
  - Klasyczne zabezpieczenie przeciwko ROP
  - Obecny na systemach od Windows Vista wzwyż
  - Obrazy wykonywalne muszą mieć ustawioną flagę kompatybilności (w trakcie kompilacji)
  - Internet Explorer działa w trybie randomizacji adresów od wersji 8



# User-mode exploitation

- Układ pamięci wirtualnej procesu **wczoraj**

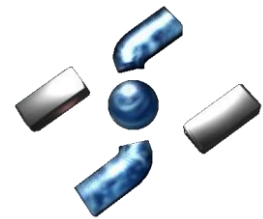
...			...
...			...
...			...
a.exe	0x00400000	0x00400000	a.exe
Internal.dll	0x00600000	0x00600000	Internal.dll
...			...
msvcrt.dll	0x75C70000	0x75C70000	msvcrt.dll
kernel32.dll	0x76E10000	0x76E10000	kernel32.dll
...			...
ntdll.dll	0x773B0000	0x773B0000	ntdll.dll
...			...
System A		System B	



# User-mode exploitation

- Układ pamięci wirtualnej procesu **dziś**

...			...
...			...
...			...
a.exe	0x00400000	0x00400000	a.exe
Internal.dll	0x00600000		...
...		0x00700000	Internal.dll
msvcrt.dll	0x75C70000		...
kernel32.dll	0x76E10000	0x75F70000	msvcrt.dll
...		0x77000000	kernel32.dll
ntdll.dll	0x773B0000		...
...		0x774F0000	ntdll.dll
System A		System B	

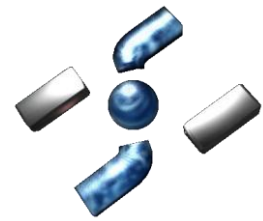


# User-mode exploitation

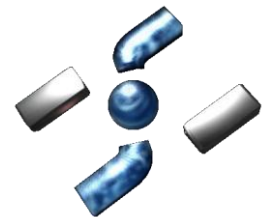
- Omijanie ASLR
  - Wyciek adresu
  - Memory spraying (*heap spraying, JIT spraying*)
  - Odszukanie niekompatybilnego modułu



# User-mode exploitation



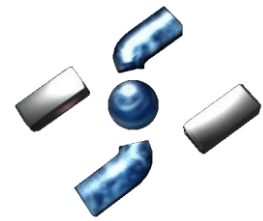
- DEP
  - Ochrona niewykonywalnych obszarów pamięci
  - Domyślnie włączony dla wszystkich 64-bitowych aplikacji
  - Działa również w trybie jądra
    - ... gdzie wszystkie moduły są 64-bitowe



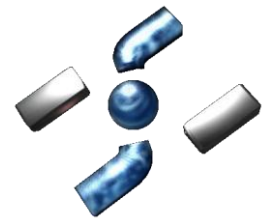
# User-mode exploitation

- Omijanie DEP
  - Najczęściej – *Return-Oriented Programming*
  - Rzadziej – JIT spraying (nowa technika)

# User-mode exploitation

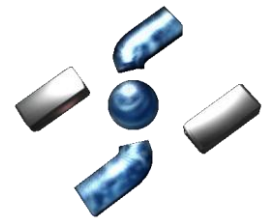


ROP?



# User-mode exploitation

- ROP (Return-Oriented Programming)
  - Znane od kilkunastu(dziesięciu?) lat
  - Wcześniej pod nazwą *ret2anything* (libc)
  - Lub „recykling kodu” 😊
  - Szczególnie użyteczne po wprowadzeniu DEP (*Data Execution Prevention*)

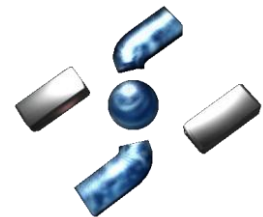


# User-mode exploitation

ROP – na czym to polega?

**Wcześniej:** uruchamiamy *payload*, umieszczony w pamięci (np. na stosie)

**Obecnie:** uruchamiamy *payload*, korzystając z gadżetów



# User-mode exploitation

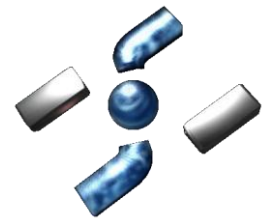
## WTF Gadżety???

Gadżet: *krótki* kod maszynowy, zakończony instrukcją skoku (najczęściej powrotu)

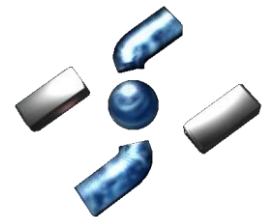
```
...  
0x0C0C0C0C: POP  EAX  
0x0C0C0C0D: POP  EBX  
0x0C0C0C0E: POP  ECX  
0x0C0C0C0F: LEAVE  
0x0C0C0C10: RET  
...
```

Przykład gadżetu

# User-mode exploitation



„Komponowanie” payloadu, poprzez tworzenie ciągu *gadżetów*, składającego się w poprawny *shellcode*

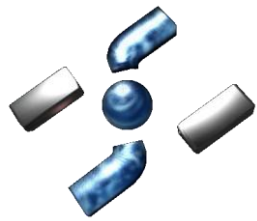


# User-mode exploitation

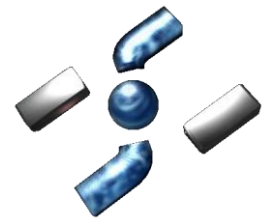
- Wystarczy, żeby zaalokować pamięć wykonywalną i skopiować tam pozostałą część *payloadu*
- Jedno ale – potrzebna znajomość adresów obrazów wykonywalnych



# User-mode exploitation



ASLR FTW 😊



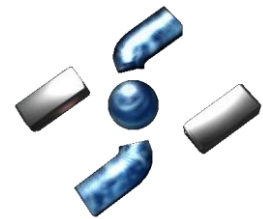
# User-mode exploitation

## Różnice pomiędzy x86 a x64 ROP

### 1. Konwencja wywołań WINAPI

#### – 32-bit: **STDCALL**

- Wszystkie argumenty kładzione na stos
- Wartość zwracana = EAX
- Funkcja wywoływana przywraca stan stosu



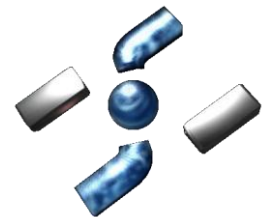
# User-mode exploitation

## Różnice pomiędzy x86 a x64 ROP

### 1. Konwencja wywołań WINAPI

#### – 64-bit: ~**FASTCALL**

- Pierwsze cztery argumenty w RCX, RDX, R8, R9; kolejne na stosie
- Rezerwacja miejsca na stosie, dla argumentów przekazanych przez rejestry

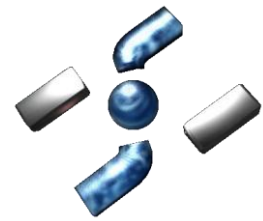


# User-mode exploitation

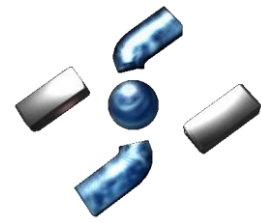
Wnioski?

... trudniej ... 😊

# User-mode exploitation

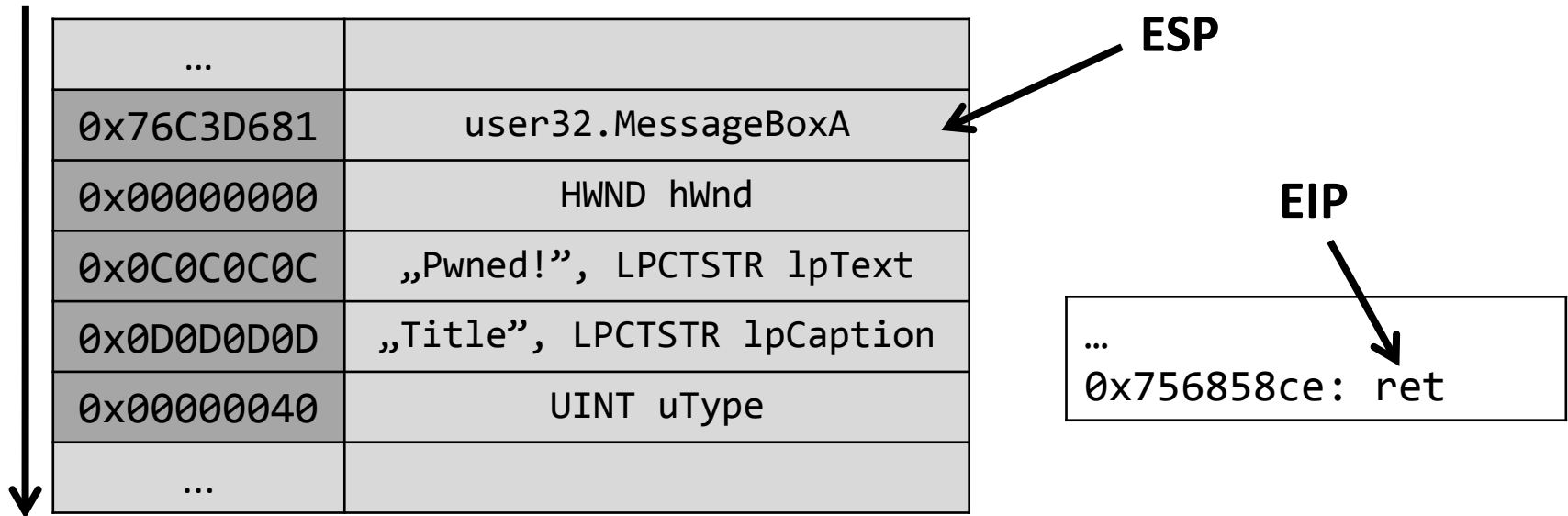


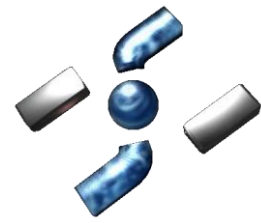
Przykład: wywołanie funkcji *MessageBoxA*,  
z kontrolowanymi argumentami



# User-mode exploitation

- 32-bit:





# User-mode exploitation

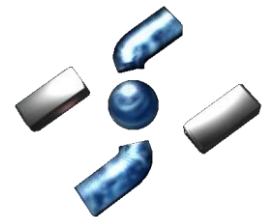
64-bit:

RSP

RIP

...  
0x756858ce: ret

...	
0x00000000`75BA1083	XOR RCX, RCX RET
0x00000000`76E1122D	POP RDX RET
0x00000000`0C0C0C0C	„Pwned!”, LPCTSTR lpText
0x00000000`773BA12E	POP R8 POP R9 RET
0x00000000`0D0D0D0D	„Title”, LPCTSTR lpCaption
0x00000000`00000040	UINT uType
0x00000000`76C3D681	user32.MessageBoxA
0x20 * '\0'	Stack padding
...	



# User-mode exploitation

## ROP a kanoniczny zapis adresów wirtualnych

0000000000000000

0

111111110000010000111000010100101100010101010000

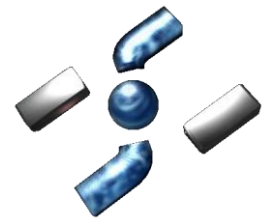
16 bitów dopełnienia

48 bitów - właściwy adres wirtualny

- W każdym adresie trybu użytkownika pojawiają się dwa zera
  - Terminator dla ciągów tekstowych ANSI i UNICODE
  - Problem w przypadku tekstowego przepełnienia na stosie

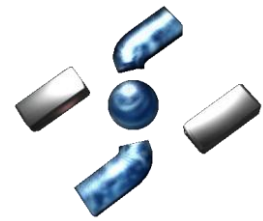


# **PRZEMYŚLENIA RÓŻNE**



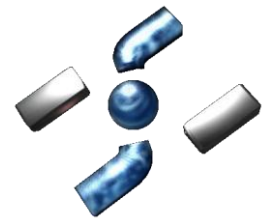
# Random stuff

- Większy rozmiar wskaźników → większe struktury wewnętrzne (nagłówki sterty itd.)
  - HEAP\_ENTRY
  - POOL\_ENTRY



# Random stuff

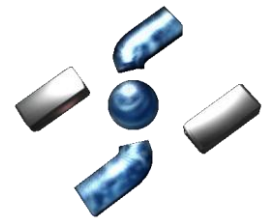
- $\text{DWORD} * \text{DWORD} \neq \text{Integer Overflow}$   
anymore 😊



# Random stuff

- Błędne założenia dot. rozmiaru typów
  - SIZE\_T
  - Wskaźniki

```
SIZE_T values[10];  
memset(values, 0, 4*10);
```

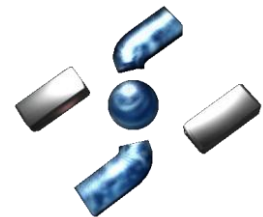


# Random stuff

- Klasa błędów *Integer Wrap*

- QWORD value = 0x12345678AB;

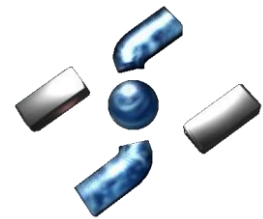
użycie (DWORD)value → utrata górnych bitów,  
możliwy buffer overflow



# x64 exploitation tools

- Windbg + 64-bit guest
  - VMWare
  - VirtualBox
- WinAppDbg
- MOSDEF x64
- IDA64, IDAPython64

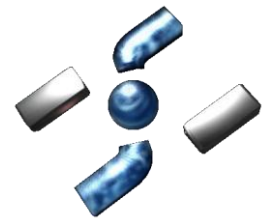
# **PODSUMOWANIE**



# Podsumowując...

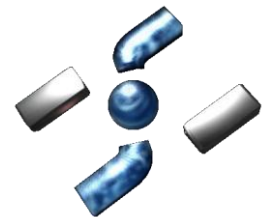
- x86-64 architekturą przyszłości
- Niezbyt dobrze zbadana
  - „*First rootkit targeting 64-bit Windows spotted in the wild*” Posted on **27.08.2010**... WHAT?
- Otwiera spore możliwości





# Podsumowując

- Poczekajmy na nowe, charakterystyczne dla x64 klasy błędów 😊
  - *CVE-2009-2847: Linux Kernel <= 2.6.31-rc5 sigtalstack 4-Byte Stack Disclosure (padding attack)*



# Pytania

Dziękuję za uwagę!

Q & A

E-mail: [j00ru.vx@gmail.com](mailto:j00ru.vx@gmail.com)

Blog: <http://j00ru.vexillum.org/>

Twitter: <http://twitter.com/j00ru>