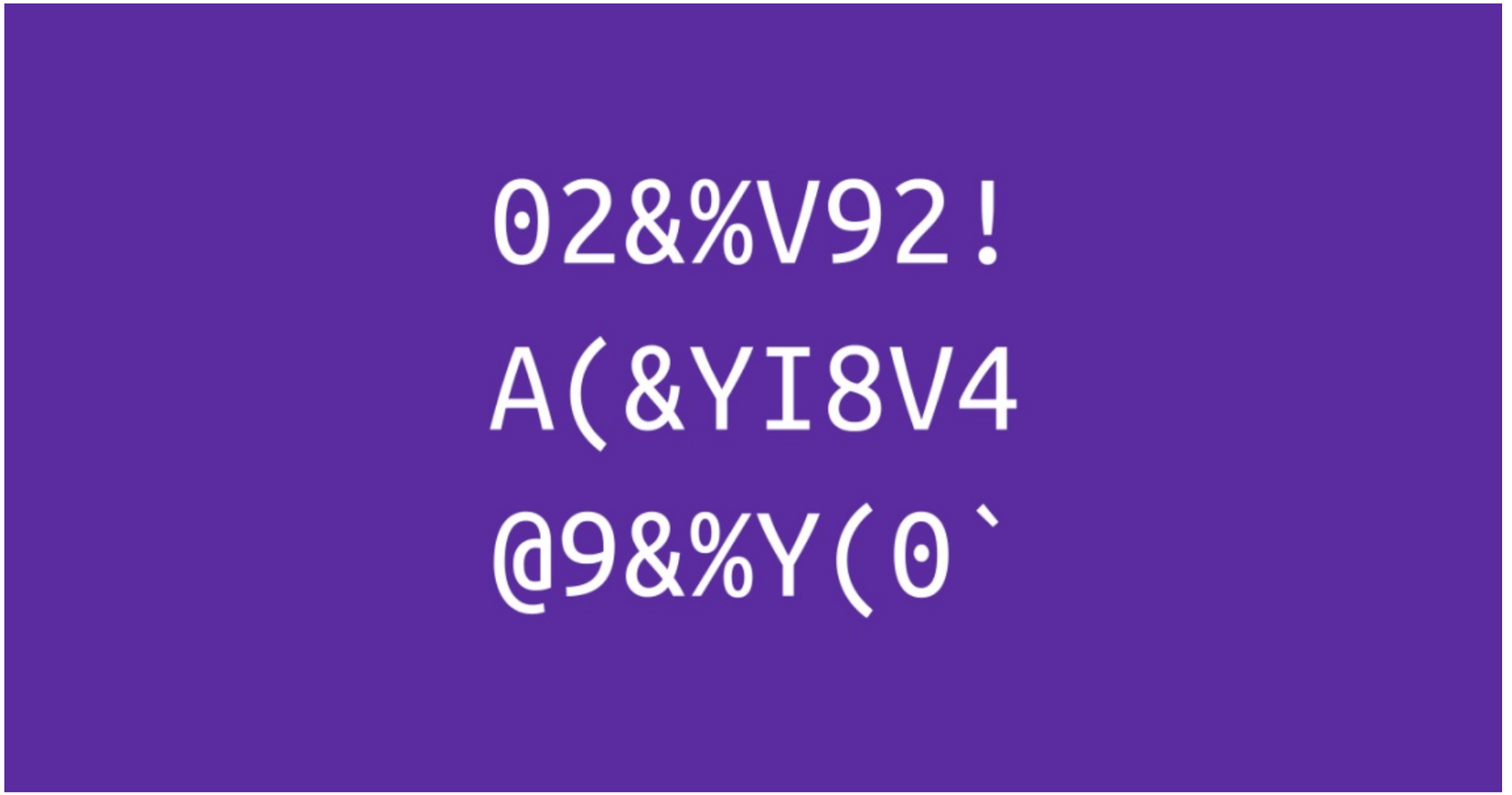

[RElog] Ukrywanie danych (string’ów) wewnątrz pliku wykownywalnego (sheadovas/artykuly/relog/relog-ukrywanie-danych-stringow-wewnatrz-pliku-wykownywalnego/)

Kwi 08, 2016 / RElog (sheadovas/category/artykuly/relog/)

Kilka słów o uniemożliwieniu wyciągnięciu np. hasła do bazy danych, z której korzysta program.

Załóżmy, że posiadamy napisaliśmy program, który z jakiegoś powodu musi posiadać jakieś wrażliwe dane (np login i hasło do bazy danych) zapisane „na sztywno” w kodzie.

Jak łatwo zdać sobie sprawę takie dane domyślnie nie są w żaden chronione i są potencjalnie wystawione na atak. W tym wpisie chciałbym się skupić na kilku potencjalnych rozwiązaniach tego problemu.

Przykładowy kod

Dla pełnego zrozumienia problemu poniżej przedstawiam uproszczony program, którego celem jest „połączenie się” z bazą danych.

Przykładowy program C++

```cpp
#include <string>
#include <iostream>

void connect()
{
    std::string username = "my_username";
    std::string password = "my_password";

    // "połącznie"
    std::cout << username << ' ' << password << "\n";
}


int main()
{
  connect();
}
```

Jak widzimy dane do logowania trzymamy jako zwykły string, przekonajmy się dlaczego nie jest to dobry pomysł:

```
$ strings my_program
...
my_username
my_password
...
```

Jak widzimy już takie proste narzędzie jak *strings* umożliwia znalezienie nam stringów przechowujących nasz login i hasło z wnętrza pliku wykonywalnego.

Rozwiązania problemu

Poniżej chciałbym przedstawić kilka potencjalnych rozwiązań problemu, które intuicyjnie przychodzą do głowy

1. Zaszyfrowanie danych.
2. Zaciemnienie sekcji danych i kodu.
3. Wymuszenie podania loginu i hasła w trakcie działania programu
4. Usunięcie bezpośredniego łączenia z bazą danych, komunikacja z bazą danych tylko przez serwer.

Dwa ostatnie rozwiązania są potencjalnie najbezpieczniejsze, a to dlatego że zakładają zrzucenie odpowiedzialności weryfikacji danych na serwer, wrażliwych danych nie ma w pliku wykonywalnym.

Jednakże nie zawsze możemy zastosować to rozwiązanie, a to dlatego że tymi danymi nie zawsze są dane służące do logowania, a np kod wykonujący jakiś sprytny algorytm, dodajmy że program w takim przypadku może pracować w trybie offline.

1. Szyfrowanie

Szyfrowanie jest rzeczą, która nasuwa się na samym początku jednakże należy pamiętać, że skoro nasz program jest w stanie dane odszyfrować, to postronna osoba po chwili analizy będzie w stanie zrozumieć nasz program i cały proces odwrócić.

Załóżmy że zaszyfrowaliśmy dane do logowania i proces odszyfrowania odbywa się w 100% po stronie programu chwilę przed logowaniem, załóżmy że teraz nie byliśmy w stanie znaleźć loginu i hasła narzędziem strings ponieważ te dane niczym się nie różniły od innych śmieci.

Spójrzmy na fragment kodu odpowiedzialnego za łączenie z bazą danych (dla zwiększenia czytelności wynik z *Hoppera*):

```
(gdb) disas connect
0000000000400f72         mov        rbp, rsp
0000000000400f75         push       rbx
0000000000400f76         sub        rsp, 0x78
0000000000400f7a         lea        rax, qword [ss:rbp+var_20]
0000000000400f7e         mov        rdi, rax
0000000000400f81         call       j__ZNSaIcEC1Ev
0000000000400f86         lea        rdx, qword [ss:rbp+var_20]
0000000000400f8a         lea        rax, qword [ss:rbp+var_80]
0000000000400f8e         mov        esi, 0x401319
0000000000400f93         mov        rdi, rax
0000000000400f96         call       j__ZNSsC1EPKcRKSaIcE
0000000000400f9b         lea        rax, qword [ss:rbp+var_20]
0000000000400f9f         mov        rdi, rax
0000000000400fa2         call       j__ZNSaIcED1Ev
0000000000400fa7         lea        rax, qword [ss:rbp+var_20]
0000000000400fab         mov        rdi, rax
0000000000400fae         call       j__ZNSaIcEC1Ev
0000000000400fb3         lea        rdx, qword [ss:rbp+var_20]
0000000000400fb7         lea        rax, qword [ss:rbp+var_70]
0000000000400fbb         mov        esi, 0x401325                               ; "EYsBq__[C@N"
0000000000400fc0         mov        rdi, rax
0000000000400fc3         call       j__ZNSsC1EPKcRKSaIcE
0000000000400fc8         lea        rax, qword [ss:rbp+var_20]
0000000000400fcc         mov        rdi, rax
0000000000400fcf         call       j__ZNSaIcED1Ev
0000000000400fd4         lea        rdx, qword [ss:rbp+var_80]
0000000000400fd8         lea        rax, qword [ss:rbp+var_60]
0000000000400fdc         mov        rsi, rdx
0000000000400fdf         mov        rdi, rax
0000000000400fe2         call       j__ZNSsC1ERKSs
0000000000400fe7         lea        rax, qword [ss:rbp+var_50]
0000000000400feb         lea        rdx, qword [ss:rbp+var_60]
0000000000400fef         mov        rsi, rdx
```

Po chwili analizy jesteśmy w stanie stwierdzić, że:

- username = „EYs]_M@DqEM”
- password = „EYsBq__[C@N”

Zauważamy również call’e do funkcji o nazwie „_Z7decryptSs”, sama nazwa łudząco przypomina wyrażenie „decrypt”, a więc spróbujmy się w niej zanurzyć aby zrozumieć jak działa.

```
(gdb) disas decrypt
```

```
Dump of assembler code for function _Z7decryptSs:
   0x0000000000400c24 <+0>: push   rbp
   0x0000000000400c25 <+1>: mov    rbp,rsp
   0x0000000000400c28 <+4>: push   rbx
   0x0000000000400c29 <+5>: sub    rsp,0x28
   0x0000000000400c2d <+9>: mov    QWORD PTR [rbp-0x28],rdi
   0x0000000000400c31 <+13>:    mov    QWORD PTR [rbp-0x30],rsi
   0x0000000000400c35 <+17>:    lea    rax,[rbp-0x15]
   0x0000000000400c39 <+21>:    mov    rdi,rax
   0x0000000000400c3c <+24>:    call   0x400a20 <_ZNSaIcEC1Ev@plt>
   0x0000000000400c41 <+29>:    lea    rdx,[rbp-0x15]
   0x0000000000400c45 <+33>:    mov    rax,QWORD PTR [rbp-0x28]
   0x0000000000400c49 <+37>:    mov    esi,0x401318
   0x0000000000400c4e <+42>:    mov    rdi,rax
   0x0000000000400c51 <+45>:    call   0x4009d0 <_ZNSsC1EPKcRKSaIcE@plt>
   0x0000000000400c56 <+50>:    lea    rax,[rbp-0x15]
   0x0000000000400c5a <+54>:    mov    rdi,rax
   0x0000000000400c5d <+57>:    call   0x400a00 <_ZNSaIcED1Ev@plt>
   0x0000000000400c62 <+62>:    mov    DWORD PTR [rbp-0x14],0x0
   0x0000000000400c69 <+69>:    jmp    0x400c9e <_Z7decryptSs+122>
   0x0000000000400c6b <+71>:    mov    eax,DWORD PTR [rbp-0x14]
   0x0000000000400c6e <+74>:    movsxd rdx,eax
   0x0000000000400c71 <+77>:    mov    rax,QWORD PTR [rbp-0x30]
   0x0000000000400c75 <+81>:    mov    rsi,rdx
   0x0000000000400c78 <+84>:    mov    rdi,rax
   0x0000000000400c7b <+87>:    call   0x4009e0 <_ZNSsixEm@plt>
   0x0000000000400c80 <+92>:    movzx  eax,BYTE PTR [rax]
   0x0000000000400c83 <+95>:    xor    eax,0x2f
   0x0000000000400c86 <+98>:    add    eax,0x3
   0x0000000000400c89 <+101>:   movsx  edx,al
   0x0000000000400c8c <+104>:   mov    rax,QWORD PTR [rbp-0x28]
   0x0000000000400c90 <+108>:   mov    esi,edx
   0x0000000000400c92 <+110>:   mov    rdi,rax
```

To co tutaj widzimy to stworzenie pustego obiektu klasy std::string oraz pętlę przechodzącą po całym stringu. Daruję sobie całą analizę i skupmy się na właściwym fragmencie „szyfrującym”:

Szyfrowanie

```
0000000000400c80        movzx     eax, byte [ds:rax]
0000000000400c83        xor       eax, 0x2f
0000000000400c86        add       eax, 0x3
```

Mamy tutaj wrzucenie poszczególnego znowu do rejestru eax, następnie zostaje on xorowany z wartością 47 (0x2f), po czym zostanie dodana liczba 3 (0x3)

Jak łatwo się domyślić cały proces jest bardzo łatwo odwrócić operacją odwrotną dla xor jest xor, a dla dodawania odejmowanie (w razie gdybyśmy chcieli odwrócić cały proces), napiszmy sobie mały program, którego zadaniem jest deszyfrowanie:

decrypt C++

```cpp
std::string decrypt(std::string word)
{
  std::string a="";
  for(int i=0;i<word.size();++i)
  {
    a += (word[i] ^ 47) + 3;
  }

  return a;
}
```

Jak się przekonamy, to jest kalka tego co widzieliśmy w postaci kodu maszynowego, otrzymane dane zgadzają się z tym co znaleźliśmy wcześniej (my_username:my_password). Szyfrowanie mogło wyglądać w ten sposób:

```
std::string encrypt(std::string word)
{
  std::string a = "";
  for(int i=0;i<word.size();++i)
  {
    a += (word[i] - 3) ^ 47;
  }
  return a;
}
```

Jak widzimy nie tylko poznaliśmy zaszyfrowane login i hasło, ale zdołaliśmy także ten proces odwrócić!

Zazwyczaj wystarczy nam jedynie znalezienie sposobu na poznanie odszyfrowanego loginu i hasła (równie dobrze mogliśmy też śledzić w debuggerze proces deszyfrowania i w ten sposób te dane wyciągnąć), największą słabością tego sposobu jest to, że kod maszynowy jest czytelny i w większości przypadków po odpowiednio długiej analizie powinniśmy być w stanie go odtworzyć.

2. Obfuskacja

> Wikipedia:
>
> **Zaciemnianie kodu** (także obfuskacja, z ang. obfuscation) to technika przekształcania programów, która zachowuje ich semantykę, ale znacząco utrudnia zrozumienie

Obfuskacja kodu

Jednym ze sposobów zaciemnienie kodu jest wykorzystanie faktu, że instrukcja MOV jest kompletna z punktu widzenia Maszyny Turinga[1] (polecam zajrzeć do tego dokumentu) to możliwe jest zastąpienie wielu innych instrukcji używając wyłącznie instrukcji MOV. Co to oznacza? Otóż dla (nie tylko) przeciętnego reversera znaczniej prostsza jest analiza kodu generowanego normalnie przez kompilator. Przykładowo poniżej widzimy zrzut funkcji main wyświetlającej „Super Secret String” (standardowe opcje gcc).

objdump

```
000000000040052d <main>:
  40052d:   55                      push   %rbp
  40052e:   48 89 e5                mov    %rsp,%rbp
  400531:   bf d4 05 40 00          mov    $0x4005d4,%edi
  400536:   e8 d5 fe ff ff          callq  400410 <puts@plt>
  40053b:   b8 00 00 00 00          mov    $0x0,%eax
  400540:   5d                      pop    %rbp
  400541:   c3                      retq
  400542:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  400549:   00 00 00
  40054c:   0f 1f 40 00             nopl   0x0(%rax)
```

Poniżej widzimy fragment robiący dokładnie to samo, skompilowany domyślnymi opcjami movcc

objdump

```
08048744 <main>:
 8048744:   a1 e8 33 3f 08          mov    0x83f33e8,%eax
 8048749:   ba 44 87 04 88          mov    $0x88048744,%edx
 804874e:   a3 70 32 1f 08          mov    %eax,0x81f3270
 8048753:   89 15 74 32 1f 08       mov    %edx,0x81f3274
 8048759:   b8 00 00 00 00          mov    $0x0,%eax
 804875e:   b9 00 00 00 00          mov    $0x0,%ecx
 8048763:   ba 00 00 00 00          mov    $0x0,%edx
 8048768:   a0 70 32 1f 08          mov    0x81f3270,%al
 804876d:   8b 0c 85 80 d8 04 08    mov    0x804d880(,%eax,4),%ecx
 8048774:   8a 15 74 32 1f 08       mov    0x81f3274,%dl
 804877a:   8a 14 11                mov    (%ecx,%edx,1),%dl
 804877d:   89 15 60 32 1f 08       mov    %edx,0x81f3260
 8048783:   a0 71 32 1f 08          mov    0x81f3271,%al
 8048788:   8b 0c 85 80 d8 04 08    mov    0x804d880(,%eax,4),%ecx
 804878f:   8a 15 75 32 1f 08       mov    0x81f3275,%dl
 8048795:   8a 14 11                mov    (%ecx,%edx,1),%dl
 8048798:   89 15 64 32 1f 08       mov    %edx,0x81f3264
 804879e:   a0 72 32 1f 08          mov    0x81f3272,%al
 80487a3:   8b 0c 85 80 d8 04 08    mov    0x804d880(,%eax,4),%ecx
 80487aa:   8a 15 76 32 1f 08       mov    0x81f3276,%dl
 80487b0:   8a 14 11                mov    (%ecx,%edx,1),%dl
 80487b3:   89 15 68 32 1f 08       mov    %edx,0x81f3268
 80487b9:   a0 73 32 1f 08          mov    0x81f3273,%al
 80487be:   8b 0c 85 80 d8 04 08    mov    0x804d880(,%eax,4),%ecx
 80487c5:   8a 15 77 32 1f 08       mov    0x81f3277,%dl
 80487cb:   8a 14 11                mov    (%ecx,%edx,1),%dl
 80487ce:   89 15 6c 32 1f 08       mov    %edx,0x81f326c
 80487d4:   a1 60 32 1f 08          mov    0x81f3260,%eax
 80487d9:   8b 15 64 32 1f 08       mov    0x81f3264,%edx
 80487df:   8b 04 85 20 a3 04 08    mov    0x804a320(,%eax,4),%eax
 80487e6:   8b 04 90                mov    (%eax,%edx,4),%eax
 80487e9:   a3 60 32 1f 08          mov    %eax,0x81f3260
```

W tym momencie ręczna analiza robi się momentalnie mało przyjemna. Jeżeli kod wykonywałby np. jakieś szyfrowanie to nam byłoby niezwykle trudno zrozumieć ten algorytm analizując wyłącznie jego kod maszynowy.

W powyższych listingach użyłem [movfuscator'a (https://github.com/xoreaxeaxeax/movfuscator)]. Narzędzia tego typu mają swoje ograniczenia (np. *movfuscator* nie ukrywa stringów, ale w tym przypadku można zastosować np system obfuskacji stringów[2]), ale w przypadku zaciemnienia jedynie fragmentów odpowiadających za jakieś magiczne obliczenia powinien się nadać i niewątpliwie wydłuży przynajmniej w nieznacznym stopniu poznanie wszystkich tajników naszego kodu o czym już wspomniałem.

VM

Innym potencjalnym sposobem na jeszcze dodatkowe zaciemnienie kodu jest utworzenie maszyny wirtualnej, która obsługiwałaby naszą własną składnię języka maszynowego. Całkiem sporo opowiedział o tym [Gynvael Coldwind (https://www.youtube.com/watch?v=wROV20w6Nt0)], w skrócie chodzi o to aby ściśle tajny program był napisany w naszym własnym języku, następnie my taki wygenerowany bajt-kod przekazujemy do naszego normalnego programu, z kolei on uruchomi maszynę wirtualną, a na niej wykona nasz algorytm.

W takim przypadku aby zrozumieć algorytm to należy najpierw zrozumieć jak działa maszyna wirtualna -> napisać disassembler -> (teraz można) zrozumieć algorytm, w dodatku przy tego typu obfuskacji nikt nie mówi, że sama składnia musi być „przyjazna", co pokazali *Gyn* i *j00ru*, a rozbił na części pierwsze [pakt (https://gdtr.wordpress.com/2011/06/24/solving-pimp-crackme-by-j00ru-and-gynvael-coldwind/)].

Podsumowanie

Jak widać istnieje wiele sposobów na utrudnienie pracy reverserowi, jeżeli możemy to powinniśmy zrzucić część którą chcemy chronić od wystawienia na widok np. na serwer, który jedynie zwróci wynik.

Chciałbym też zwrócić uwagę, że ten temat został opisany przeze mnie jedynie pobieżnie (nie poruszyłem np. tematu packerów) i na pewno nie został wyczerpany, zachęcam do samodzielnego zgłębienia tematu i zajrzenia do m.in. „Materiałów dodatkowych".

Materiały dodatkowe

- [1][„mov is Turing complete” (https://www.cl.cam.ac.uk/~sd601/papers/mov.pdf)]
- [2][Przykładowy system obfuskacji stringów (http://www.codeproject.com/Articles/502283/Strings-Obfuscation-System)]
- [3][Dissect || PE (http://research.dissect.pe/docs/blackhat2012-paper.pdf)]

Code ON!