Writeup Watchmen – CONFidence 2019 Teaser

Niedawno – po roku przerwy – odbył się kolejny CTF z marką CONFidence. Tym razem zmienił się organizator – zamiast drużyny Dragon Sector[1] (który zajmuje się organizowaniem Dragon CTF[2] na konferencji PWNing[3]) konkurs został przygotowany przez zespół p4[4]. Na zawodników czekało 16 zadań z różnych kategorii. Najliczniej reprezentowana była kategoria „reverse engineering" i to na zadaniu właśnie z tej kategorii skupimy się w poniższym artykule.[5]

#	team	points	Sanity check	The Lottery	My admin panel	Web 50	Go Machine	Watchmen	Pudliszki	Oldschool	Elementary	Really Suspicious ...	Bro, do you even l...	Count me in!	pdfmt	Neuralflag	Sloik	Game server
1.	hxp	2722	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑
2.	Balsn	2365	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	
3.	Dragon Sector	2116	⚑		⚑	⚑	⚑	⚑		⚑	⚑	⚑	⚑	⚑	⚑		⚑	⚑
4.	Made In MIM	2000	⚑	⚑	⚑		⚑		⚑	⚑	⚑	⚑	⚑	⚑		⚑	⚑	⚑
5.	OpenToAll	1966	⚑	⚑	⚑	⚑			⚑	⚑	⚑	⚑	⚑	⚑	⚑		⚑	⚑
6.	justCatTheFish	1850	⚑	⚑	⚑	⚑			⚑	⚑	⚑	⚑	⚑	⚑		⚑	⚑	⚑
7.	217	1668	⚑		⚑		⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑	⚑		
8.	RPISEC	1624	⚑	⚑	⚑	⚑		⚑	⚑	⚑	⚑			⚑	⚑	⚑		
9.	HackingForSoju	1520	⚑		⚑	⚑	⚑	⚑			⚑		⚑	⚑		⚑		⚑
10.	Magnum	1507	⚑		⚑	⚑	⚑		⚑	⚑	⚑	⚑	⚑	⚑		⚑	⚑	

CTF	Teaser CONFidence CTF 2019
Waga CTFtime.org	24.95 (https://ctftime.org/event/760)
Liczba drużyn (z niezerową liczbą punktów)	546
System punktacji zadań	Od prostych (51) do trudnych (357)
Liczba zadań	16
Podium	1. hxp (Niemcy) – 2722 pkt. 3. Balsn (Tajwan) – 2365 pkt. 3. Dragon Sector (Polska) – 2116 pkt.
Zadanie	Watchmen

O ZADANIU

> *Who watches the watchmen?*
> *watchmen.tar.gz 44.4 kB*

To enigmatyczne zdanie jest całym opisem zadania „watchmen". Zadanie to okazało się być jednym z trudniejszych na CTFie – ukończyło go jedynie 13 spośród 546 zarejestrowanych zespołów.

Rozwiązywanie zaczynamy od pobrania i wypakowania paczki z zadaniem (ciągle dostępnej na stronie https://confidence2019.p4.team/challenge/watchmen) – Listing 1.

Listing 1. Standardowe kroki po pobraniu zadania na CTF

```
$ sha256sum watchmen_601cc9(...)5438.tar.gz
601cc9(...)5438  watchmen_601cc9(...)5438.tar.gz
$ 7z x watchmen_601cc9(...)5438.tar.gz
$ 7z x watchmen_601cc9(...)5438.tar
$ file watchmen.exe
watchmen.exe: PE32 executable (console) Intel 80386 (stripped to
external PDB), for MS Windows
```

Na pierwszy rzut oka wydaje się, że stoimy przed typowym windowsowym crackme. Po uruchomieniu zadanie „prosi" o podanie flagi, po czym weryfikuje, czy jest ona poprawna (Listing 2):

Listing 2. Testowe uruchomienie

```
> ./watchmen.exe
Once you realize what a joke everything is, being the Comedian is
the only thing that makes sense.
p4{czy_to_poprawna_flaga?}
No. Not even in the face of Armageddon. Never compromise
```

Naszym zadaniem jest, jak zwykle, znalezienie poprawnej flagi (czyli takiej, która zostanie zaakceptowana przez program).

WSTĘPNE ROZPOZNANIE

Jak to zwykle w zadaniach z kategorii Reverse Engineering, dobrym pomysłem na start jest otworzenie binarki w deasemblerze[6]. Do niedawna deasembler w tym kontekście był prawie że synonimem „IDA Pro"[7], ale niedawno zostało to zmienione przez NSA, które udostęp-

1. https://ctftime.org/team/3329
2. https://ctftime.org/ctf/103
3. https://www.instytutpwn.pl/konferencja/pwning/
4. https://ctftime.org/team/5152
5. Dla pełnej przejrzystości dodam, że niżej podpisany jest również członkiem zespołu p4, a także twórcą opisywanego zadania.
6. Nie jest to oczywiście jedyna opcja. Inny dobry pomysł to rozpoczęcie analizy od strony dynamicznej i śledzenie zdarzeń systemowych powodowanych przez aplikację. Można w ten sposób bardzo szybko nabyć ogólne pojęcie o programie, a czasami nawet rozwiązać zadanie blackboxowo.
7. IDA Pro to najpopularniejszy deasembler, do niedawna nie mający praktycznie konkurencji w świecie profesjonalnych reverserów.

Zapraszamy na autorskie szkolenia
z zakresu bezpieczeństwa IT

{ Bezpieczeństwo aplikacji WWW }

{ Offensive HTML, SVG, CSS and other Browser-Evil }

{ Wprowadzenie do bezpieczeństwa IT }

{ Szkolenie przygotowujące do egzaminu CEH }
(Certified Ethical Hacker)

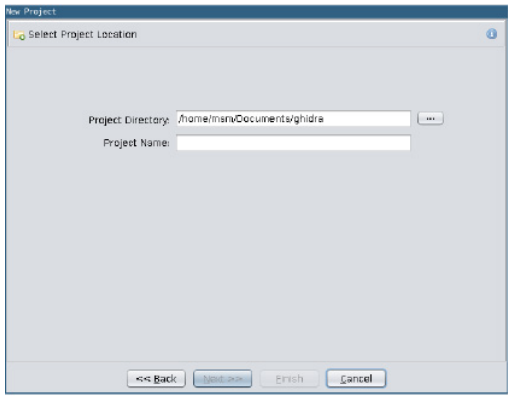
www.securitum.pl/oferta/szkolenia

Patroni medialni: sekurak.pl rozwal.to

niło konkurencyjne narzędzie Ghidra[8], które działa zaskakująco dobrze i może (moim zdaniem) w wielu przypadkach zastąpić Idę. Jednym z większych atutów Ghidry jest jej cena, a w zasadzie jej brak, bo została ona udostępniona za darmo[9] (co również ciekawe w przypadku tak specjalistycznego narzędzia). Z tego powodu zdecydowałem się napisać ten artykuł, korzystając wyłącznie z jej pomocy. Pomimo że samo zadanie jest trudne, spróbowałem umieścić w artykule wystarczająco szczegółowe opisy, by nawet czytelnicy zaczynający przygodę z reversowaniem mogli za nim podążać. Szczególnie te osoby zachęcam do pobrania plików zadania i spróbowania swoich sił.

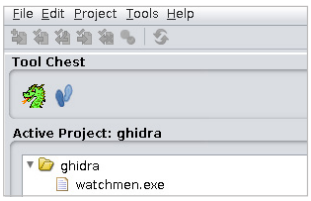
Sama Ghidra jest napisana w Javie, co trochę widać. Nie mówię tu nawet o samym UI (które nie jest piękne), ale głównie o ilości kliknięć, które trzeba wykonać, żeby rozpocząć analizę.

Zaczynamy od stworzenia projektu (lub otworzenia istniejącego). Można tego dokonać, wybierając w menu opcję *File->New Project* albo skrótem Ctrl+N (Rysunek 1).



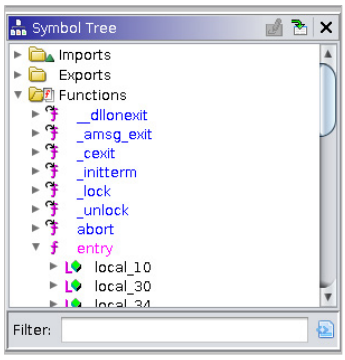
Rysunek 1. Okno tworzenia nowego projektu. Nazwę i ścieżkę projektu można wybrać dowolnie

Następnie należy zaimportować do projektu plik, który chcemy analizować – czyli w tym przypadku program watchmen.exe. Ponownie mamy do wyboru menu *File->Import* albo skrót klawiaturowy „i". W tym momencie nasz program został już załadowany do projektu (Rysunek 2).



Rysunek 2. Program watchmen.exe załadowany do projektu Ghidra

Nareszcie możemy zacząć prawdziwą pracę. Po podwójnym kliknięciu na nazwę programu otwiera się deasembler. Automatyczna analiza w Ghidra jest uboższa niż w IDA Pro i funkcja `main` nie została automatycznie wykryta, więc reversowanie musimy rozpocząć od *entry-pointa*[10]. Pomoże nam w tym okno Symbol Tree, w którym wybieramy `entry` (Rysunek 3).



Rysunek 3. Okno podglądu symboli

Jedną z ciekawszych własności Ghidry jest to, że domyślnie pokazuje ona jednocześnie widok dekompilacji i deasemblacji. Często jest to przydatne przy niskopoziomowej analizie, ale na razie dekompilacja w zupełności nam wystarczy. Spójrzmy na funkcję `entry` (Listing 3).

Listing 3. Bardzo przycięty kod funkcji entry

```
void entry(void)
{
  char cVar1;
  uint32_t uVar2;
  // (...)
  byte local_34;
  ushort local_30;
  iVar9 = 0x11;
  puVar15 = local_60;
  while (iVar9 != 0) {
        iVar9 = iVar9 + -1;
        *puVar15 = 0;
        puVar15 = puVar15 + 1;
  }
  // (...)
  DAT_0043c6a0 = DAT_0043c6a0 ^
                 (uint)(DAT_0043c6a0 == 0) *
                 (DAT_0043c6a0 ^ *(uint *)(*(int *)(in_FS_OFFSET
                 + 0x18) + 4));
  // (...)
  DAT_0043c00c = FUN_00401530(*(uint32_t *)(ppvVar11 + -1));
  if (DAT_0043c008 != 0) {
    if (DAT_0043c004 == 0) {
    ppvVar11[-2] = 0x401405;
    _cexit(*(uint32_t *)(ppvVar11 + -2));
    }
    return DAT_0043c00c;
  }
  ppvVar11[-1] = DAT_0043c00c;
  ppvVar11[-2] = 0x4014b3;
  exit((int)ppvVar11[-1]);
}
```

Została ona znacznie ograniczona w celu demonstracji (pełna dekompilacja tylko tej jednej funkcji ma ponad 200 linijek kodu!). Na szczęście wcale nie musimy jej dokładnie analizować. Po pierwsze, wiemy, że głównym zadaniem entry pointa jest przygotowanie środowiska przed wykonaniem właściwego `main` – to oznacza, że `main` będzie jedną z ostatnich wywoływanych funkcji. Sugeruje to funkcję `FUN_00401530` w powyższym listingu. Drugą rzeczą, która nam może tutaj pomóc, jest doświadczenie – prezentowana tutaj funkcja `entry` jest standardowym prologiem kompilatora gcc, więc wytrawnemu analitykowi jeden rzut oka pozwoli przeskoczyć ten etap.

8. http://ghidra-sre.org/
9. Podobno ma też być w pełni open-source. Niestety, na razie nie cały kod został opublikowany.
10. Entry point (często w asemblerze oznaczany jako funkcja `start`) to kod, który wykonuje się jeszcze przed funkcją `main` i jest generowany przez kompilator.

Zmieńmy więc nazwę z `FUN_00401530` na `main` (*ppm->Rename Function* albo skrót klawiszowy „L") i prowadźmy dalszą analizę (Listing 4).

Listing 4. Funkcja main analizowanego programu

```
int main(uint8_t param_1) {
  local_c = &param_1;
  FUN_00435ec0();
  FUN_00402138();
  local_18 = (*DAT_0043c404)(0,1,"DYNAMIC_EXEC");
  local_1c = (*DAT_0043c408)();
  local_14 = (uint)(local_18 == 0);
  if (local_1c == 0xb7) {
    FUN_00401e69();
  } else {
    if (local_1c == 0) {
      FUN_004015f8(&local_2c);
      FUN_00401700(local_2c,local_28);
      (*DAT_0043c40c)(local_18);
    }
    else {
      local_14 = 1;
    }
```

PO NITCE DO KŁĘBKA

Uwagę zwraca tutaj wywołanie kilku funkcji przez wskaźniki zamiast importowanie ich bezpośrednio. Zostało to tak zaimplementowane z powodów technicznych, ale można to też potraktować jako słabą obfuskację. Skoro te wskaźniki są wywoływane, to coś musi wcześniej ustawić ich wartość. Musi się to dziać przed pierwszym dynamicznym wywołaniem, więc nie ma wiele miejsc, w których odpowiedzialna funkcja może się chować. I rzeczywiście – robi to już funkcja `FUN_00402138` – zmieńmy jej nazwę na `LoadDynamicImports` i przeanalizujmy (Listing 5).

Listing 5. Kod funkcji ładującej dynamicznie funkcje

```
void LoadDynamicImports(void)

{
  HMODULE hModule;
  HMODULE hModule_00;

  hModule = LoadLibraryA("msvcrt.dll");
  hModule_00 = LoadLibraryA("kernel32.dll");
  _DAT_0043c3e0 = GetProcAddress(hModule,"printf");
  DAT_0043c3e4 = GetProcAddress(hModule,"memset");
  DAT_0043c3e8 = GetProcAddress(hModule,"bsearch");
  DAT_0043c3ec = GetProcAddress(hModule_00,"WaitForDebugEvent");
  DAT_0043c3f0 = GetProcAddress(hModule_00,"ContinueDebugEvent");
  DAT_0043c3f4 = GetProcAddress(hModule_00,"GetThreadContext");
  DAT_0043c3f8 = GetProcAddress(hModule_00,"SetThreadContext");
  DAT_0043c3fc = GetProcAddress(hModule_00,"ReadProcessMemory");
  DAT_0043c400 = GetProcAddress(hModule_00,"WriteProcessMemory");
  DAT_0043c404 = GetProcAddress(hModule_00,"CreateMutexA");
  DAT_0043c408 = GetProcAddress(hModule_00,"GetLastError");
  DAT_0043c40c = GetProcAddress(hModule_00,"ReleaseMutex");
  DAT_0043c410 = GetProcAddress(hModule_00,"GetModuleFileNameA");
  DAT_0043c414 = GetProcAddress(hModule_00,"CreateProcessA");
  return;
}
```

`LoadDynamicImports` sama w sobie nie jest w żaden sposób obfuskowana i wyraźnie widać, jak ładuje potrzebne sobie funkcje do zmiennych. Trochę niepokojące są funkcje ładowane z biblioteki kernel32 – `WaitForDebugEvent` oraz `Get/SetThreadContext` to funkcje charakterystyczne bardziej dla debuggerów niż dla crackme. Pozostaje nam jedynie zmienić nazwy zmiennych na bardziej odpowiednie (znowu skrótem klawiaturowym „L") i spojrzeć ponownie na kod `main` (Listing 6).

Listing 6. Kod funkcji main staje się znacznie bardziej przejrzysty (nazwy zmiennych dodane przez niżej podpisanego)

```
hMutex = (*CreateMutexA)(0,1,"DYNAMIC_EXEC");
dwLastError = (*GetLastError)();
error = hMutex == 0;
if (dwLastError == 0xb7) {
  FUN_00401e69();
}
else {
  if (dwLastError == 0) {
    FUN_004015f8(&local_2c);
    FUN_00401700(local_2c,local_28);
    (*ReleaseMutex)(hMutex);
  }
  else {
    error = true;
  }
}
```

Co tutaj właściwie się dzieje? To bardziej pytanie ze znajomości WinAPI i programowania równoległego niż z RE. Kod najpierw próbuje utworzyć mutex o nazwie `DYNAMIC_EXEC`, a później w zależności od wyniku wchodzi w odpowiednią gałąź ifa. Mutex[11] to konstrukt synchronizacyjny, którego wyróżniającą cechą jest to (w uproszczeniu), że w danym systemie może istnieć tylko jeden mutex z daną nazwą. Inaczej mówiąc, jeśli przed uruchomieniem programu watchmen.exe inny proces stworzył już mutex `DYNAMIC_EXEC`, próba utworzenia nie uda się, `GetLastError` zwróci błąd `ERROR_INVALID_HANDLE` (`0xb7` w kodzie) i wykona się kod w gałęzi `if`. W przeciwnym razie wykona się kod spod `else`.

Ale skąd w zasadzie miałby się pojawić taki mutex w systemie? Wrócimy do tego za chwilę, na razie popatrzmy w kod `FUN_00401e69` (Listing 7).

Listing 7. Kod funkcji FUN_00401e69 – coś tu jest nie tak...

```
                    FUN_00401e69
00401e69 0f 0b      UD2
00401e6b 31         ??          31h     1
00401e6c 9b         ??          9Bh
00401e6d 2a         ??          2Ah     *
00401e6e 56         ??          56h     V
00401e6f 57         ??          57h     W
00401e70 12         ??          12h
00401e71 7a         ??          7Ah     z
00401e72 b0         ??          B0h
00401e73 72         ??          72h     r
00401e74 15         ??          15h
00401e75 fe         ??          FEh
00401e76 36         ??          36h     6
00401e77 4d         ??          4Dh     M
00401e78 4f         ??          4Fh     O
00401e79 79         ??          79h     y
00401e7a 12         ??          12h
00401e7b 0f         ??          0Fh
00401e7c 0b         ??          0Bh
```

Hmm. Dekompilator niestety odmawia współpracy, a deasemblacja wygląda równie źle. Widoczna jest jedynie pierwsza instrukcja (`UD2`). W dodatku `UD2` jest instrukcją zarezerwowaną jako niepoprawna[12]. Próba wykonania jej skończyłaby się gwarantowanym wyjątkiem procesora (`#UD`, *invalid opcode exception*)[13]. Wygląda na to, że nie tędy droga – tego kodu na pewno nic nie wykona w takiej postaci. Być może gałąź `else` w funkcji `main` ma odpowiedzi na nasze pytania. Wywołuje ona po kolei funkcje `FUN_004015f8` i `FUN_00401700`. Popatrzmy na pierwszą z nich (Listing 8).

11. Od *mutual exclusion*, wzajemne wykluczenie.
12. Jeśli o tym pomyśleć dłużej, brzmi to dziwnie – opkod, dla którego Intel gwarantuje, że zawsze pozostanie niepoprawnym.
13. https://www.felixcloutier.com/x86/ud

Listing 8. Pierwsza z funkcji wołanych w gałęzi else

```
uint32_t *FUN_004015f8(uint32_t *param_1) {
  (*memset)(startupInfo,0,0x44);
  startupInfo[0] = 0x44;
  success = (*GetModuleFileNameA)(0,hModName,0x104);
  if (success == 0) { exit(1); }
  success = (*CreateProcessA)(
    hModName,0,0,0,0,2,0,0,startupInfo,&procInfo
  );
  if (success == 0) { exit(1); }
  return procInfo;
}
```

Aha! Widać jak na dłoni, że program tu najpierw pobiera swoją nazwę pliku, a później tworzy proces spod tej ścieżki (czyli uruchamia sam siebie). Warto jednak przyjrzeć się dokładnie flagom przekazanym do `CreateProcessA`. Tylko jedna z nich jest niezerowa, a okazuje się, że jest całkiem interesująca[14]. Wartość `2` oznacza `DEBUG_ONLY_THIS_PROCESS` – czyli proces uruchamia sam siebie jako debugger (proces jest początkowo wstrzymany).

Łatwo się domyślić, że druga funkcja z gałęzi `else` jest odpowiedzialna za właściwe debugowanie. Przyjrzyjmy się jej (Listing 9).

Listing 9. Druga z funkcji wołanych w gałęzi else – okazuje się być tylko cienkim wrapperem

```
void FUN_00401700(uint32_t param_1, uint32_t param_2) {
  FUN_004020c4();
  FUN_00401720(param_1, param_2);
}

void FUN_004020c4(void) {
  DAT_0043c690 = DAT_00439000;
  DAT_0043c694 = PTR_PTR_DAT_00439004;
}
```

Hmm, niewiele. W pierwszej wywoływanej funkcji (`FUN_004020c4`) też niewiele. Idźmy więc dalej (Listing 10).

Listing 10. Główna pętla debuggera

```
void FUN_00401720(uint32_t param_1, uint32_t param_2) {
  int local_70;
  uint32_t local_6c;
  uint32_t local_68;
  int isDone = 0;
  (*memset)(&local_70, 0, 0x60);
  while (local_10 == 0) {
    (*WaitForDebugEvent)(&local_70, 0xffffffff);
    if (local_70 == 1) {
      FUN_004017bf(param_1, param_2, (int)&local_70);
    }
    else {
      if (local_70 == 5) {
        isDone = 1;
      }
    }
    (*ContinueDebugEvent)(local_6c, local_68, 0x10002);
  }
}
```

Ponownie nasze przewidywania były słuszne – ta funkcja wygląda wyraźnie jak główna pętla debuggera. Konkretnie, w nieskończonej pętli nasłuchuje na zdarzenia i reaguje na dwie sytuacje (ignorując inne):

- Zdarzenie `5`, czyli `EXIT_PROCESS_DEBUG_EVENT`. Ustawia wtedy flagę `isDone` na `1`, co powoduje zakończenie programu.
- Zdarzenie `1`, czyli `EXCEPTION_DEBUG_EVENT`. Przekazuje wtedy informacje o zdarzeniu do funkcji `FUN_004017bf` (jest to struktura typu `DEBUG_EVENT`, ale Ghidra nie poradziła sobie z propagacją typów).

Nie pozostaje nam nic innego, jak kontynuować tę wycieczkę w dół króliczej nory i popatrzeć na `FUN_004017bf` (Listing 11).

Listing 11. Obsługa eventów debuggera. Pierwsza linijka została zmieniona dla czytelności (ponownie, wina słabego wsparcia typów w Ghidrze – przypisania zostały zdekompilowane jako pętla while kopiująca bajty)

```
void FUN_004017bf(uint32_t param_1, uint32_t param_2,int param_3)
{
  EXCEPTION_DEBUG_INFO info = event->u.Exception; // changed
  address = (int *)(param_3 + 0xc);
  if (info.ExceptionRecord.ExceptionCode == 0x80000004) {
    FUN_00401836(param_1,param_2,local_58);
  }
  else if (info.ExceptionRecord.ExceptionCode == 0xC000001D) {
    FUN_0040198c(param_1, param_2, address);
  }
}
```

PRZEKLĘTE DEBUGGERY, JAK ONE DZIAŁAJĄ?

Jesteśmy coraz bliżej rozwiązania zagadki. Najpierw sprawdzamy typy wyjątków obsługiwane przez kod. Niestety oficjalna dokumentacja często nie lubi podawać wartości enumów, więc zamiast tego w pierwszych wynikach znajdujemy link do kodu na githubie[15]. Tak czy inaczej, okazuje się, że `0x80000004` oznacza `EXCEPTION_SINGLE_STEP`, a `0xC000001D` oznacza `EXCEPTION_ILLEGAL_INSTRUCTION`. Hmm, *illegal instruction* – brzmi znajomo. I faktycznie, jeśli wrócimy do `FUN_00401e69`, to znajdziemy opcode `UD2` – którego jedynym celem jest wywoływanie właśnie tego wyjątku. *Single-stepping* z kolei to funkcja procesora (na x86) albo po prostu technika debugowania polegająca na wykonywaniu instrukcji pojedynczo. Po włączeniu tego trybu, po każdej kolejnej wykonanej instrukcji procesor zgłasza wyjątek `#DB` (Debug Exception), który z kolei mapuje się na znany nam już `EXCEPTION_SINGLE_STEP`.

Mamy już wszystkie klocki, możemy zastanowić się, jaki jest cel tej układanki. Z jednej strony mamy kod zaczynający się od niepoprawnej instrukcji, z drugiej debugger potrafiący obsłużyć taką sytuację oraz śledzić programy krok po kroku. Narzuca się hipoteza, że nasz debugger-rodzic czeka, aż jego „dziecko" wykona niepoprawną instrukcję, żeby zacząć go śledzić. Wtedy będzie w jakiś sposób podmieniał kod na poprawne (oryginalne) instrukcje, które zostaną bez problemu wykonane przez proces dziecka. W dodatku skoro reaguje na *single step*, jest spora szansa, że deszyfrowanie będzie działało na bardzo małych fragmentach kodu – być może nawet na pojedynczych instrukcjach. Inaczej mówiąc, mamy do czynienia z protektorem deszyfrującym kod w locie. Przeanalizujmy go, zachowując poczynione założenia.

Funkcja obsługi niepoprawnej instrukcji wygląda tak (Listing 12):

Listing 12. Funkcja obsługi niepoprawnej instrukcji

```
void __cdecl FUN_0040198c(
    uint32_t param_1, uint32_t param_2, uint32_t address) {
  FUN_0040205f(param_1,(uint32_t *)&DAT_0043c43c);
  FUN_0040205f(param_1,&DAT_0043c420);
  IsTracing = 1;
  local_10 = (uint32_t *)FUN_004020f1(address);
  local_14 = local_10[1];
  local_18 = *local_10;
  FUN_00401f9d((int)&local_14, 2);
  FUN_00401fb8(&local_3c, param_1, address, &local_14, 2);
  memcpy(DAT_0043c420, local_3c, 28);
  FUN_00401b2b(param_1, param_2, address);
  FUN_00401a86(param_2, 1);
}
```

14. https://docs.microsoft.com/en-us/windows/desktop/procthread/process-creation-flags

15. https://github.com/gdbinit/pydbg64/blob/master/MacOSX/macdll/Exception.c

Zgodnie z naszym założeniem funkcja ta powinna wykonać kilka czynności:

» Oznaczyć gdzieś, że od teraz będziemy śledzić kod krok po kroku.
» Przed użyciem protektora, na miejscu instrukcji `UD2` były inne opcody. Wypadałoby je przywrócić.
» Finalnie należy zdeszyfrować oryginalny kod.

Jako flaga do śledzenia pasuje tylko jedna zmienna – zmieniłem jej nazwę na `IsTracing`.

Funkcję, która szuka oryginalnych opcodów, też łatwo znaleźć – wynik zależy tylko od adresu, więc jest tylko jedno pasujące wywołanie (Listing 13).

Listing 13. Wyszukiwanie patcha do podanego adresu w programie

```
uint32_t __cdecl FUN_004020f1(uint32 address) {
  uint32_t uVar1;
  uint32_t local_18[5];
  local_18[0] = address;
  return (*bsearch)(
    local_18, DAT_0043c694, DAT_0043c690, 8, &LAB_004020de
  );
}
```

Jak widać, sprowadza się to do prostego wywołania systemowej funkcji `bsearch` (*binary search*, służącej do szybkiego wyszukiwania w posortowanych danych). Tablica z odpowiednimi danymi została wcześniej dostarczona przez protektor.

Ostatecznie mamy kilka funkcji, które są dobrymi kandydatami na funkcje szyfrujące. Po przejrzeniu ich wszystkich wyłania się jeden pewny kandydat (Listing 14).

Listing 14. Funkcja podejrzanie przypominająca proste szyfrowanie… Nazwy zmiennych zmienione dla czytelności

```
void decrypt(char *data, int nbytes) {
  uint32_t key[17];
  int i = 0;
  while (i < nbytes) {
    data[i] = key[local_c] - (data[i] - key[i]);
    i = i + 1;
  }
}
```

Jest to bardzo prosta, symetryczna funkcja szyfrująca. Dane są transformowane przez „odbijanie" ich od klucza (nazwanego tutaj `key`). Pozostaje tylko pytanie... Jaka jest zawartość klucza?

Tym razem to nie obfuskacja, a ponownie problem dekompilatora. Nie umieścił on inicjalizacji zmiennej w wynikowym kodzie, więc jesteśmy zmuszeni odzyskać ją bezpośrednio z kodu asemblera (Listing 15).

Listing 15. W celu odzyskania zawartości klucza jesteśmy zmuszeni spojrzeć na surowy asembler

```
MOV  dword ptr [EBP + key[0]], 0xf
MOV  dword ptr [EBP + key[1]], 0xb
MOV  dword ptr [EBP + key[2]], 0x4f
MOV  dword ptr [EBP + key[3]], 0x3e
MOV  dword ptr [EBP + key[4]], 0x89
MOV  dword ptr [EBP + key[5]], 0xac
MOV  dword ptr [EBP + key[6]], 0xff
MOV  dword ptr [EBP + key[7]], 0x81
MOV  dword ptr [EBP + key[8]], 0xba
MOV  dword ptr [EBP + key[9]], 0x7e
MOV  dword ptr [EBP + key[10]], 0xec
MOV  dword ptr [EBP + key[11]], 0xcc
MOV  dword ptr [EBP + key[12]], 0x66
MOV  dword ptr [EBP + key[13]], 0x29
MOV  dword ptr [EBP + key[14]], 0xee
MOV  dword ptr [EBP + key[15]], 0x10
```

I… to w zasadzie wszystko. Wystarczy teraz napisać własny program, który prześledzi działanie kodu od pierwszego `UD2`, deszyfrując kod i podążając za wszystkimi napotkanymi skokami. Niestety wyjdzie z tego dość sporo kodu, a samo rozwiązanie będzie trudne w implementacji i wymagało deasemblera. Jest to oczywiście poprawne rozwiązanie (to jedno z dwóch przewidzianych rozwiązań wzorcowych), ale może da się prościej…

WATCHING THE WATCHERS

Jeśli się zastanowić, każdy bajt chronionego programu musi zostać odszyfrowany chociaż na chwilę (musi się w końcu kiedyś wykonać). Może wystarczy się wstrzelić w ten krótki czas i zapisać gdzieś oryginalną wartość każdego bajta? I rzeczywiście, okazuje się to być wykonalne. W tym celu musimy najpierw znaleźć w protektorze funkcję, która wstrzykuje dane do dziecka, ale to nietrudne, kiedy wiemy, czego szukamy (Listing 16).

Listing 16. Funkcja nadpisująca dane

```
uint32_t *FUN_00401fb8(
    uint32_t *out,
    uint32_t handle,
    uint32_t address,
    uint8_t *data,
    uint32_t length,
) {
  undefined4 local_20[4];
  (*ReadProcessMemory)(handle,address,&local_20,length,0);
  (*WriteProcessMemory)(handle,address,data,length,0);
  out[0] = length;
  out[1] = adddress;
  out[2] = local_20[0];
  out[3] = local_20[1];
  out[4] = local_20[2];
  out[5] = local_20[3];
  out[6] = 1;
  return out;
}
```

Jak widzimy, funkcja `FUN_00401fb8` najpierw czyta dane do jakiejś lokalnej struktury, a później nadpisuje je w procesie podanym patchem. To pasuje idealnie do funkcji, której szukamy. To znaczy, że jeśli zaczniemy debugować debugujący proces[16] i „wepniemy się" w jakieś miejsce w tej funkcji, to wystarczy wypisywać gdzieś wstrzykiwane dane oraz ich adres. Można to zrobić za pomocą prostego skryptowania akcji breakpointa w debuggerze, albo nawet hookowania funkcji. Liczy się wynik:

```
Patch at 401e69 c9 8d
Patch at 401e69 55 95 a9 bd 94 10 b1 0e 08 28 96 55 ca b0 73 0a
Patch at 401e6a 95
Patch at 401e6a 89 e5 1b c4 e2 7b 26 d0 3a 78 b1 62 42 43 3e ab
Patch at 401e6c 83 ec 48 f9 16 c0 2e 82 f9 00 60 d9 7e ff ce bb
Patch at 401e6f c7 04 24 cc a0 43 00 52 9f f0 1d 8c 9b b7 ee e9
Patch at 401e76 e8 19 50 03 00 d5 23 64 9d e8 fa de 8d 34 9a 05
Patch at 436e94 1f f1
Patch at 436e94 ff 25 f0 d1 43 00 b0 a4 29 a5 26 91 75 b0 28 6c
Once you realize what a joke everything is, being the Comedian is t
Patch at 401e7b 91 d1
Patch at 401e7b 8d 45 d6 37 d6 c0 f6 67 f6 7c 77 4d 45 84 f2 c4
Patch at 401e7e 89 44 24 04 63 d0 1a c7 0f 71 24 a0 c0 f9 75 8a
Patch at 401e82 c7 04 24 2f a1 43 00 52 ba f1 1d 8c 9b b7 f3 8d
Patch at 401e89 e8 fe 4f 03 00 d5 23 69 41 28 15 78 b0 58 ed 06
Patch at 436e8c 1f f1
Patch at 436e8c ff 25 f4 d1 43 00 b0 a4 39 8b e8 6b 01 08 28 6c
p4{is_this_flag}
Patch at 401e8e 7d 66
Patch at 401e8e a1 b0 d1 43 00 5b 2a 16 3e e3 ce df 48 31 2b 80
Patch at 401e93 89 04 24 d8 d1 94 f7 22 f3 0d f2 71 42 00 f2 38
Patch at 401e96 e8 49 50 03 00 d5 23 64 9d 28 fa 9c c2 c1 9d cb
Patch at 436ee4 1f f1
Patch at 436ee4 ff 25 c8 d1 43 00 b0 a4 29 a5 fe 91 75 b0 28 6c
Patch at 401e9b 91 d1
Patch at 401e9b 8d 45 d6 37 16 c0 b4 9c 83 7f 3d eb 3b 2d 5a 45
```

Rysunek 4. W wyniku uruchomienia skryptu otrzymujemy pierwsze poprawne bajty na każdym offsecie

16. Tu skojarzenie/inspiracja do nazwy zadania.

Nie widać tego bardzo dobrze, ale w tym momencie mamy na talerzu każdą instrukcję co najmniej raz. Pomijając krótkie zapisy (artefakt pracy protektora), pierwszy patch zaczyna się od `55`, a drugi od `89 e5`. Te bajty deasemblują się do `push ebp` i `mov ebp, esp`[17], co upewnia nas w przekonaniu, że mamy już poprawnie zdeszyfrowane dane.

Po uruchomieniu takiego skryptu dla całego programu sortujemy wyniki po adresie (żeby wyciągnąć poprawną długość opkodów), wyciągamy surowe bajty i zapisujemy je z powrotem do oryginalnego pliku. Jeśli wszystko poszło dobrze, po dekompilacji nowo otrzymanego „czystego" programu otrzymamy oryginalny `main` (Listing 17).

Listing 17. Szczęśliwie odzyskany kod

```
uint32_t FUN_00401e69(void) {
  uint uVar1;
  char local_2e [33];
  char local_d;

  puts("Once you realize what a joke everything is, being the
        Comedian is the only thing that makes sense.");
  scanf("%32s",local_2e);
  fflush((FILE *)_iob_exref);
  uVar1 = FUN_00401e30(local_2e);
  local_d = (char)uVar1;
  if (local_d == 0) {
    puts("No. Not even in the face of Armageddon.
          Never compromise");
  }
  else {
    puts("What happened to the American Dream? It came true!
          You\'re lookin\' at it.");
  }
  return 0;
}
```

OSTATNIE STARCIE

W tym momencie zadanie z trudnego RE staje się bardzo prostym (zasługującym pewnie na 100 punktów). Algorytm walidacji (Listing 18) szyfruje dane podane przez użytkownika za pomocą własnego wymyślonego szyfru blokowego (Listing 19, Listing 20).

Listing 18. Standardowe sprawdzenie hasła w CTFac, czyli porównanie transformowanych danych z zapisaną na sztywno stałą

```
uint __cdecl FUN_00401e30(char *param_1) {
  FUN_00401e09(param_1);
  return memcmp(&DAT_0043a0a8,param_1,0x20) == 0;
}
```

Listing 19. Własna funkcja szyfrująca, powtarzająca funkcję rundy 16 razy

```
void __cdecl FUN_00401e09(char *param_1) {
  int local_8 = 0;
  while (local_8 < 0x10) {
    FUN_00401ddf(param_1);
    local_8 = local_8 + 1;
  }
}
```

Listing 20. Funkcja rundy, typowa dla szyfru blokowego

```
void __cdecl FUN_00401ddf(char *param_1) {
  XorBytes(param_1);
  RotateBits(param_1);
  PermuteBytes(param_1);
}
```

Każda z operacji w rundzie jest łatwo odwracalna:

- Xor jest operacją trywialnie odwracalną (sam jest swoją odwrotnością).
- `RotateBits` traktuje szyfrowane dane jako jedną wielką 256-bitową liczbę i rotuje ją bitowo o 4 miejsca w lewo. Odwrotność to oczywiście rotowanie o 4 miejsca w prawo.
- `PermuteBytes` wykonuje ustaloną permutację danych i zwraca ją. Dla każdej permutacji istnieje permutacja odwrotna, więc tu również nie ma problemów.

Z tego wynika, że w celu rozwiązania zadania wystarczyło napisać odwrotności tych wszystkich operacji, a następnie złożyć je ze sobą w odpowiedniej kolejności. Cała ta praca, żeby móc zdeszyfrować flagę i otrzymać pochwałę od programu (Listing 21).

Listing 21. Warto było: rozwiązaliśmy zadanie

```
> ./program.exe
Once you realize what a joke everything is, being the Comedian is
the only thing that makes sense.
p4{~JusticeIsComingToAllOfUs...}
What happened to the American Dream? It came true! You're lookin'
at it.
```

PRZEMYŚLENIA KOŃCOWE

Zadanie „watchmen" stawiało przed uczestnikami konkursu niemałe wyzwanie: wymagało znajomości API Windowsa, obycia w inżynierii wstecznej oraz umiejętności skryptowania debuggera (rozwiązanie dynamiczne) albo dobrego programowania niskopoziomowego (rozwiązanie statyczne). Z drugiej strony dawało się rozwiązać na wiele sposobów oraz nie stawiało sztucznych utrudnień – kodu było dość niewiele i nie krył się z tym, co robi. W sam raz wpasowało się w naszą kategorię „trudniejszych" zadań.

Drugą kwestią, którą chcę poruszyć w zakończeniu, są moje mieszane uczucia co do Ghidry. Z jednej strony działa wyjątkowo dobrze jak na narzędzie, które wyskoczyło prawie że znikąd. Nadaje się też moim zdaniem do wykorzystania w pracy na co dzień. Z drugiej strony dekompilacja jest gorszej jakości niż ta w IDA Pro. Szczególnie cierpi propagacja typów, a sam kod wynikowy okazjonalnie pomija krytyczne informacje. Zachęcam każdego do wyrobienia sobie ostatecznej opinii samodzielnie.

Jarosław Jedynak

Rozwiązanie zadania *Watchmen* zostało nadesłane przez zespół p4, polski zespół CTF-owy, który jest królem CTFów jak lew jest królem dżungli: https://ctftime.org/team/5152

17. `push ebp; mov ebp, esp` to najbardziej popularny prolog funkcji na x86.