Writeup Gothic – CONFidence 2019 Finals

W jednym z poprzednich numerów opisywaliśmy zadanie Watchmen z CTF na konferencji Confidence, który po roku przerwy odbył się kolejny raz. Tym razem zmienił się organizator – zamiast drużyny Dragon Sector (która zajmuje się organizowaniem Dragon CTF na konferencji PWNing) konkurs został opracowany przez zespół p4. Na zawodników czekało 18 zadań, od bardzo prostych do całkiem trudnych. Jak to zwykle na zawodach organizowanych przez p4 bywa, najwięcej zadań (aż 6) należało do kategorii „reverse engineering". W tym numerze omówimy dwa zadania opierające się na, odpowiednio zmodowanej, kultowej w Polsce grze Gothic w wersji Demo.

22 teams total

Place	Team	CTF points	Rating points
1	Dragon Sector	6203.000	50.000
2	hxp	4749.000	31.640
3	EmpireCr0wn	3446.000	22.222
4	Made In MIM	3446.000	20.138
5	bruNOOOOO	3420.000	18.784
6	justCatTheFish_	3420.000	17.950
7	qxh	2756.000	14.679
8	_hxp_	2230.000	12.113
9	hxpwn	2092.000	11.209
10	KonungarnirFróðleikar	1675.000	9.251

CTF	CONFidence CTF 2019 Finals
Waga CTFtime.org	25 (https://ctftime.org/event/823)
Liczba drużyn (z niezerową liczbą punktów)	22
System punktacji zadań	Od prostych (51) do trudnych (357)
Liczba zadań	18
Podium	1. Dragon Sector (Polska) – 6203 pkt. 3. hxp (Niemcy) – 4749 pkt. 3. EmpireCr0wn (Wielka Brytania) – 3446 pkt.
Zadanie	Gothic RE oraz Gothic Crypto

O ZADANIU

Zadanie składało się z dwóch części, ale obie korzystały z tej samej binarki. Decyzję, którą część chcemy rozwiązywać, podejmowaliśmy już w głównym menu gry:

Rysunek 1. Menu główne zmodyfikowanej gry Gothic

Obie części były w dużej mierze niezależne – tak naprawdę do części kryptograficznej gra była tylko ciekawym dodatkiem. Zacznijmy więc od omówienia mniej standardowego zadania – Gothic RE.

ZADANIE GOTHIC RE

- *Works with wine or VMware Workstation Player. A custom dubbing from p4 team included. A few hints:*
- *Hold E and press W/S/A/D in order to interact with the environment (chest, NPCs, items).*
- *In the RE task, you don't need to do anything besides talking to the initial NPC.*
- *Flag consists of p4{ prefix, uppercase hex bytes and } suffix.*

Gra zaczyna się tak jak na Gothica przystało – rozmową z Diego (Rysunek 2.1).

Rysunek 2.1. Standardowa rozmowa z Diego na początek gry

Szybko jednak konwersacja zbacza na dość... dziwny tor. Bezimienny przekonuje swojego rozmówcę, że zna sekretny kod, który zniszczy barierę i uwolni więźniów (Rysunek 2.2).

Rysunek 2.2. Niestandardowa opcja dialogowa

A następnie, wcielając się w gracza, podajemy po kolei kolejne znaki sekretnego kodu. Do wyboru są cyfry 0-9, znaki A-F oraz znak } kończący flagę (Rysunek 2.3). Łatwo się domyślić, że musimy wprowadzić jakiś szesnastkowy sekret. Niestety, w grze nie ma żadnych podpowiedzi, o jaki kod może chodzić. Oznacza to, że musimy zacząć reversować kod gry.

Rysunek 2.3. Opcje dialogowe, które mamy do wyboru

Jedną z trudniejszych części zadania jest odkrycie, które funkcje w programie są odpowiedzialne za obsługę wpisywania flagi. Nie ma na to jednego dobrego sposobu – można było użyć metod dynamicznych (ale bardzo łatwo się zgubić w gąszczu wszystkich funkcji wołanych przez grę) albo statycznych. Wybór padł na te drugie. Szukając ciekawych napisów w binarce, udało się znaleźć funkcję rejestrującą interesujące callbacki (Listing 1).

Listing 1. Sygnatura interesującej funkcji.

```
; int __cdecl Script::registerExternalsSpacer(ZStringSpacer *)
public Script::registerExternalsSpacer(CParser *)
Script::registerExternalsSpacer(CParser *) proc near
```

Ciekawe wydały się zdarzenia `gtl_setState`, `gtl_getState` oraz `gtl_validate`. W funkcji `registerExternalsSpacer` kilkukrotnie powtarza się fragment rejestrujący handlery do niektórych zdarzeń (Listing 2.1).

Listing 2.1. Rejestracja getStateCallback do obsługi zdarzenia gtl_getState

```
mov     ebx, Parser::DefineExternalSpacer
lea     eax, [ebp+var_58]
mov     dword ptr [esp], offset aGtlGetstate ; "gtl_getState"
mov     ecx, eax
call    ZStringSpacer::ZStringSpacer(char const*)
sub     esp, 4
mov     dword ptr [esp+10h], 0
mov     dword ptr [esp+0Ch], 2
mov     dword ptr [esp+8], offset getStateCallback
lea     eax, [ebp+var_58]
mov     [esp+4], eax    ; char *
mov     eax, [ebp+arg_0]
mov     [esp], eax      ; this
call    ebx ; Parser::DefineExternalSpacer
```

Obsługa zdarzenia `gtl_setState` jest bardzo prosta – po prostu czyta ona zmienną `ds:Script::state` i zwraca ją do wywołującego za pomocą funkcji `Parser::SetReturnInt`, która znajduje się w głównym module (tzn. gothic.exe), przez co nie mamy jej w symbolach.

```
getStateCallback proc near
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, ds:Script::state
mov     [esp], eax
mov     ecx, 903E70h
mov     eax, 716F90h
call    eax
sub     esp, 4
mov     eax, 1
leave
retn
getStateCallback endp
```

Funkcja `validate` sprawdza jedynie, czy wpisaliśmy już 40 znaków oraz czy pewna flaga w pamięci gry jest zapalona. Najciekawszym zdarzeniem jest więc `setState` (Listing 2.2).

Listing 2.2. Początek funkcji setStateCallback (nazwa nadana już podczas analizy)

```
setStateCallback proc near
push    ebp
mov     ebp, esp
sub     esp, 8
mov     eax, offset suspiciousFunction
push    eax
push    large dword ptr fs:0
mov     large fs:0, esp
call    setStateImpl
mov     eax, [esp]
```

```
mov     large fs:0, eax
add     esp, 8
mov     eax, 1
leave
retn
setStateCallback endp
```

Najpierw robi on jakieś operacje z rejestrem `fs` (do nich jeszcze wrócimy), a później woła kolejną funkcję (nazwaną tutaj `setStateImpl`). Jej kod jest długi w asemblerze, ale łatwo wyrazić go w paru linijkach C (Listing 2.3):

Listing 2.3. Początek funkcji setStateCallback (nazwa nadana już podczas analizy)

```
void setStateImpl() {
     int inp_state;
     // Parser::GetParameterInt z silnika gry
     getPressedCharacter(Parser::instance, &inp_state);
     if (steps[step] != inp_state) {
         valid = false;
     }
     states[step] = inp_state;
     if (steps[step]) { step++; }
     state = inp_state ^ 0xDACEFABE;
     writeToMemory(0x7F58A4, &inp_state);
 }
```

Jak widać, przy każdym naciśnięciu klawisza gra wykonuje operację xor i sprawdza, czy nowo otrzymany stan zgadza się z tym oczekiwanym zapisanym w tablicy `state` (Listing 2.4)

Listing 2.4. Wartości stanu oczekiwane przez program

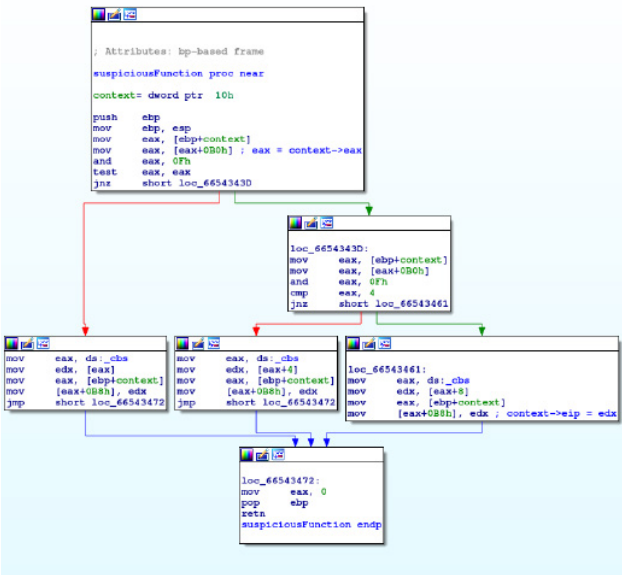
```
dd 0FF04B27Fh, 50BE628h, 0F880FDBDh, 0FF612A75h, 0FF6AD402h
dd 0FF19FE98h, 0FF914F3Bh, 11F97BAh, 0FFB618D1h, 0FEF3DA71h
dd 0FFFDF2DDh, 0FEE0B0BFh, 0FF398CEDh, 74A4E1Fh, 0FE081EBCh
dd 0FF3D41D9h, 0FFCEDEB3h, 1179FDDh, 0FCC090Fh, 200F5CAh
dd 0FD9A60B0h, 107E4BBh, 0F806FFE0h, 1AEC320h, 0FC162ACDh
dd 8E6096h, 0CD8ED1h, 49D9A2h, 0FEFF50AEh, 0FF3CB2BCh
dd 7086EF4h, 0FE501F1Ch, 0FF10F09Fh, 0FFB728D7h, 0FEF41C46h
dd 0FFFBF7D7h, 0FEE13E95h, 0DD875Fh, 5997C0h, 0DB1BA1h
```

W tym momencie wystarczy skopiować funkcję walidującą oraz tablicę stanu i odzyskać klawisze, których się spodziewa. A przynajmniej tak się wydaje... Okazuje się, że w praktyce tak odzyskany przez nas tajny kod nie działa. Zmyliło to większość (albo nawet wszystkich) graczy próbujących rozwiązać zadanie, ale okazuje się, że zostaliśmy wpędzeni w ślepą uliczkę – funkcja, którą przeanalizowaliśmy, wcale nie sprawdza poprawności flagi.

Po krótkiej chwili desperacji/złości/zaskoczenia ponownie rozpoczynamy poszukiwania właściwej funkcji. Kluczowa jest ostatnia linijka funkcji `setStateImpl`, którą łatwo przeoczyć jako nic ważnego. Dokonuje ona zapisu jakiejś wartości pod adres `0x7F58A4`. Okazuje się, że pamięć pod tym adresem jest tylko do odczytu i próba zapisu zawsze kończy się wyjątkiem… Dlaczego w takim razie gra dalej działa?

I tutaj wracamy znowu do Listingu 2.2. Osobom obeznanym z niskopoziomowymi detalami działania Windowsa na pewno rzucił się on w oczy. Zapis pod adres `fs:0` to prawie pewny znak, że mamy do czynienia z mechanizmem SEH[1] – zwłaszcza że są w to zamieszane wskaźniki na funkcje. A co to takiego, ten SEH?

Czy zastanawialiście się kiedyś, jak działają np. konstrukcje typu try/catch w językach programowania? W jaki sposób taki C++ może dowiedzieć się, że nastąpiło na przykład dzielenie przez zero albo właśnie złe odwołanie do pamięci? Jeśli się nad tym zastanowić, język musi mieć jakieś wsparcie systemu operacyjnego. W przypadku Windowsa zostało to zrealizowane właśnie przez SEH[2]. Dokładny opis zasad działania SEH nie mieści się w ramach tego artykułu, ale nie jest też potrzebny do rozwiązania zadania – wystarczy wiedzieć, że system zakłada, iż pamięć pod adresem `fs:0` wskazuje na początek listy handlerów i w razie wyjątku są one wykonywane po kolei. W naszym przypadku łatwo zauważyć, że handlerem w Listingu 2.2 może być tylko `suspiciousFunction`. Popatrzmy więc na niego (Rysunek 2.4).



Rysunek 2.4. Kod obsługi wyjątku

Parametr, który dostaje handler `SEH`, to tzw. `CONTEXT`[3] – zapisany stan procesora w momencie wyjątku. Zależnie od wartości rejestru `eax` w momencie wyjątku wykonanie programu jest kontynuowane w innym miejscu programu. W języku C++ można by to przedstawić następująco (Listing 2.5):

Listing 2.5. Kod obsługi wyjątku w C++

```
if ((Context->Eax & 0xF) == 0) {
    Context->Eip = cbs->a;
} else if ((Context->Eax & 0xF) == 0x4) {
    Context->Eip = cbs->b;
} else {
    Context->Eip = cbs->c;
}
```

Po analizie dochodzimy do wniosku, że w przypadku, który nas interesuje, wykona się funkcja `sub_665E455D`. A tam… trafiamy na funkcję bardzo podobną do `setStateImpl`, ale korzystającej z innej tabeli stanu (Listing 2.6). Wystarczy więc uruchomić nasz solver jeszcze raz, ale tym razem dla poprawnych danych, żeby otrzymać poprawną flagę:

1. https://docs.microsoft.com/en-us/windows/win32/debug/structured-exception-handling

2. Tak naprawdę to dużo bardziej skomplikowane. Już w Windowsie XP została wprowadzona alternatywna metoda (tzw. VEH), która obiecywała większą wydajność (brak narzutu w runtime, zakładając optymistyczną ścieżkę wykonania), nie przyjęta jednak gorąco przez kompilatory. A wersja 64-bitowa systemu jeszcze bardziej skomplikowała sprawę, bo kolejny raz zupełnie zmieniła podejście do obsługi wyjątków, porzucając poprzednie mechanizmy na rzecz struktur `RUNTIME_FUNCTION` w metadanych pliku PE.

3. https://www-user.tu-chemnitz.de/~heha/viewchm.php/hs/Win32SEH.chm/

Listing 2.6. Prawdziwe dane

```
dd 0F3492493h, 10089B17h, 0FFD70058h, 0FC4A516Dh, 7D1FC9h
dd 0B55FC4h, 0FE6D37EBh, 0BCFAE2h, 0FDF296DBh, 0F2DBAB4Eh
dd 0FD7F89B5h, 0FF8FD4CDh, 0FE3CBC49h, 0FA518645h, 0FE578396h
dd 0F5E87222h, 288C4C7h, 6EAD127h, 0EEF07375h, 0FF8E13A9h
dd 0FF3FA66Eh, 0FF14E7A0h, 0FF696076h, 0FF00AB99h, 61297Eh
dd 104A8D2h, 0F0C6114h, 12BD26Eh, 0FF28363Eh, 0FFE2513Bh
dd 0FF2BCD79h, 6625774h, 0FF5EF283h, 0FF8D3C46h, 0FFC7A521h
dd 0FECEB5EAh, 0EE4D152Ch, 137D9Fh, 349B611h, 0FF9FCAE3h
```

Otrzymujemy w ten sposób poszukiwane hasło:

```
p4{6733746869635F6833783072}.
```

ZADANIE GOTHIC CRYPTO

» *Do you think it's possible to break a random RSA encrypted message? If you somehow manage, there is a chest in our Gothic game, with a flag waiting for you. Rest of the challenge info is in the game as well.*
» *Works with wine or VMware Workstation Player. A custom dubbing from p4 team included.*
» *A few hints:*
» *Hold E and press W/S/A/D in order to interact with the environment (chest, NPCs, items).*
» *In the crypto task, you don't need to go anywhere outside the initial room in order to complete the task.*
» *The annoying NPC may have some interesting items.*

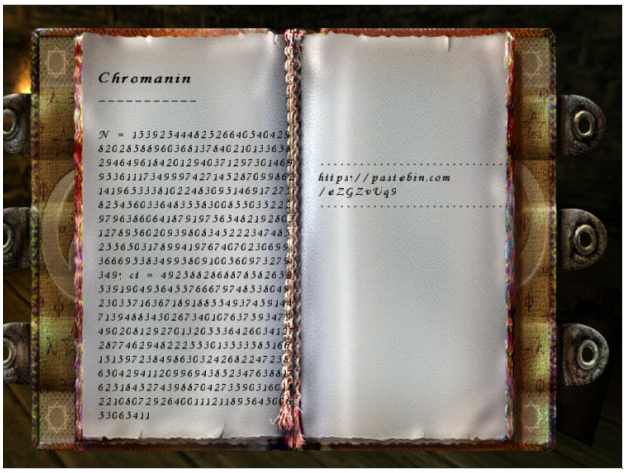
Po wejściu do gry z opcji „Play Crypto" (z technicznego punktu widzenia jest to po prostu wczytanie stanu gry przygotowanego przez twórcę zadania) rzucają nam się w oczy dwa elementy wystroju pokoju. Po pierwsze, skrzynia stojąca pod ścianą (Rysunek 3). Po zalogowaniu do gry nasz wzrok jest skierowany prosto na nią, więc można się domyślić, że ma jakiś związek z zadaniem. Jest też wspomniana w opisie zadania (*there is a chest in our Gothic game, with a flag waiting for you*, w wolnym tłumaczeniu: „w grze jest skrzynia z czekającą na ciebie flagą"). Dodatkowo weterani gry Gothic od razu zauważą, że nie zachowuje się jak zwykła skrzynka w grze – gracz może próbować otworzyć ją wytrychem, ale kod do zamka jest bardzo długi (na aż 46 znaków), a sam wytrych łamie się dopiero po wpisaniu całego kodu[4]. Odpada więc zgadywanie kodu w grze – musimy go jakoś zdobyć.

Rysunek 3. Podejrzana skrzynia stojąca pod ścianą

4. W oryginalnym Gothicu wytrych łamał się zaraz po pierwszym błędnym ruchu, dało się więc łamać kody do skrzynek metodą siłową „bit po bicie".

Tu wchodzi do gry drugi rzucający się obiekt w pokoju – NPC o imieniu Mud (w polskiej wersji Wrzód) – Rysunek 3.1. Trudno go nie zauważyć, bo odzywa się do nas co chwilę, nie dając nam nawet wytchnienia. Już po 30 sekundach zaczyna być denerwujący (mimo bardzo ładnego dubbingu podłożonego przez zawodniczkę z p4 – a może właśnie ze względu na niego). Nie pozostawia wątpliwości, że to o nim wspomina opis zadania (*The annoying NPC may have some interesting items*, „denerwujący NPC może mieć jakieś ciekawe przedmioty"). Po kilku minutach większość graczy albo zasugeruje się tą podpowiedzią, albo podąży za swoim naturalnym instynktem i pobije niewinnego Muda, żeby zdobyć jego przedmioty. Jest to dobry trop. Zdobywamy w ten sposób księgę „Chromanin" (Rysunek 3.2) z treścią zagadki. Na szczęście nie trzeba przepisywać całego `N` oraz ciphertextu (uff) – można je znaleźć po wejściu w link podany na prawej karcie[5], razem z kodem użytym do ich wygenerowania.

Rysunek 3.1. Natarczywa postać uparcie nas zagadująca



Rysunek 3.2. Księga „Chromanin" z treścią zagadki

Kolejny dzień, kolejny atak na RSA

Po wejściu pod podany link na pastebinie znajdujemy następujący kod:

Listing 3. Kod dostarczony przez autora zadania

```
def main():
    secret_bits = 46
    challenges = [generate_challenge(secret_bits) for _ in
range(10)]
```

5. https://pastebin.com/eZGZvUq9

```
    solvable = filter(lambda (N, ct, M, is_solvable): is_
solvable, challenges)
    assert len(solvable) > 0  # just a hint that it's a common
property
    N, ct, M, is_easy = solvable[0]
    print("N = " + str(int(N)))
    print("ct = " + str(int(ct)))

def generate_challenge(secret_bits):
    N, e = generate_RSA_pubkey()
    M, is_easy = generate_secret(secret_bits)
    ct = pow(M, e, N)
    return N, ct, M, is_easy

def generate_RSA_pubkey():
    modulus_bitsize = 1024
    e = 65537
    p = getStrongPrime(modulus_bitsize / 2)
    q = getStrongPrime(modulus_bitsize / 2)
    N = p * q
    return N, e

def generate_secret(secret_bits):
    M = random.randint(2 ** (secret_bits - 1), 2 ** secret_bits - 1)
    import solver
    return M, solver.is_solvable(M, secret_bits)

main()
```

Jak to zwykle w zadaniach z kryptografii bywa, nic nie rzuca się na pierwszy rzut oka jako oczywisty błąd. W zadaniu generowane jest kilka „wyzwań" (challenge) i zwracane pierwsze, które spełnia pewną nieznaną własność (`is_easy`). Jest to podpowiedź autora zadania, że niektóre przypadki czymś się wyróżniają – ale też, że szansa na taki przypadek jest dość wysoka (skoro występuje wśród 10 pierwszych wylosowanych wyzwań).

Każde wyzwanie składa się z klucza publicznego (wygenerowanego w dość standardowy sposób) oraz sekretu (liczby z przedziału od 2^{s-1} do 2^s-1, dla s=46). Wynik szyfrowania sekretu `M` za pomocą klucza publicznego (`e, N`) nazywamy `ct`.

Jak widać, stoi przed nami spore wyzwanie. W księdze z gry znajduje się tylko `N` i `ct`, więc gracz musi odszyfrować `ct`, znając tylko klucz publiczny (`e, N`) – inaczej mówiąc, połamać RSA. Zadanie wydaje się nierozwiązywalne, ale w końcu to CTF – autor raczej nie wymaga od nas niemożliwego, w kodzie musi być błąd, który można wykorzystać.

Skupmy się na sekrecie `M` – co może w nim być takiego specjalnego. Po pierwsze – jest dość mały. 2^{46} to dużo, ale nie bardzo dużo – jest to nawet w zasięgu ataku *brute-force* dla organizacji dysponującej bardzo dużymi zasobami obliczeniowymi (np. przysłowiowego NSA). Atak wyglądałby mniej więcej tak[6]:

Listing 3.1. Atak brute-force

```
N, ct = challenge_data
for M in range(2 ** 45, 2 ** 46 - 1):
    next_ct = pow(M, e, N)
    if next_ct == ct:
        print "Found M: ", M
```

Niestety, nie każdy gracz CTF dysponuje prywatnym klastrem obliczeniowym porównywalnym z tym NSA, więc trzeba myśleć dalej. Jaka więc może być słabość w `M`? Tu z pomocą przychodzi doświadczenie: w przypadku RSA zazwyczaj problemy pojawiają się, kiedy liczby rozkładają się na czynniki pierwsze. I rzeczywiście – skoro `M` jest dowolną dużą losową liczbą, istnieje szansa, że rozkłada się na dwa duże czynniki (niekoniecznie pierwsze!). Załóżmy więc, że `M` jest iloczynem dwóch liczb (nazwijmy je `m1` i `m2`):

```
M = m1 * m2
```

W takiej sytuacji możemy przeprowadzić pewnego rodzaju atak typu *Meet In The Middle* („spotkania w środku")[7]. Dla uproszczenia załóżmy, że `m1` i `m2` są bardzo podobnej wielkości – powiedzmy, że obie mają pomiędzy 22 a 25 bitów[8]. Możemy wtedy przeprowadzić nasz atak *brute-force* w trochę bardziej okrężny sposób:

Listing 3.2. Inne podejście do ataku brute-force

```
N, ct = challenge_data
for m1 in range(2 ** 22, 2 ** 25):
    for m2 in range(2 ** 22, 2 ** 25):
        M = m1 * m2
        next_ct = pow(M, e, N)
        if next_ct == ct:
            print "Found M: ", M
```

Po co tak komplikować rozwiązanie? Jeszcze tego nie widać, ale za pomocą prostego przekształcenia matematycznego możemy dokonać wielkiej optymalizacji (wszystkie operacje modulo N):

$$ct \equiv M^e$$

$$ct \equiv (m_1 * m_2)^e$$

$$ct \equiv m_1^e * m_2^e$$

$$ct * m_2^{-e} \equiv m_1^e$$

Kluczowa jest ostatnia linijka – możemy sprowadzić szukanie `M` do szukania takich m_1 i m_2, że ct * m_2^{-e} = m_1^e. Ostatni przykładowy kod przed ostatecznym rozwiązaniem wyglądałby tak:[9]

Listing 3.3. Rozpisanie ataku brute-force na dwie części

```
N, ct = challenge_data
for m1 in range(2 ** 22, 2 ** 25):
    for m2 in range(2 ** 22, 2 ** 25):
        m2_inv = modinv(m2, N)
        left = ct * pow(m2_inv, e, N)
        right = pow(m1, e, N)
        if left == right:
            print "Found M: ", m1 * m2
```

Ciągle mamy dwie zagnieżdżone pętle, ale jesteśmy bardzo blisko pozbycia się ich. Wystarczy zauważyć, że lewa strona równania zależy tylko od `m2`, a prawa tylko od `m1`... Na myśl przychodzi prekomputacja – możemy najpierw obliczyć pętlę tylko dla lewej strony równania, później tylko dla prawej, a następnie porównać wyniki i zobaczyć, kiedy wyszło nam to samo. Wersja poglądowa wygląda tak:

Listing 3.4. Atak meet in the middle

```
for m1 in range(2 ** 22, 2 ** 25):
    right = pow(m1, e, N)
    right_results[right] = m1
```

6. Poza tym, że taki hipotetyczny atak musiałby być zrównoleglony na tysiące rdzeni, żeby mógł skończyć się w sensownym czasie.

7. Nie mylić z atakiem „Man In The Middle", który również jest często skracany do MITM.

8. Co zgadzałoby się z danymi z zadania – wynik mnożenia dwóch liczb ma mniej więcej tyle samo bitów, co suma bitów mnożonych liczb – czyli [10 bitowa liczba] * [10 bitowa liczba] ~= [20 bitowa liczba].

9. Operację `modinv` można w Pythonie przeprowadzić na wiele sposobów – np. korzystając z kodu na https://stackoverflow.com/questions/4798654/modular-multiplicative-inverse-function-in-python, albo z funkcji `invert` z biblioteki gmpy2.

testerzy.pl

BEST POLISH COMMERCIAL CONFERENCE 2018

CONFERENCE

WORKSHOPS

DELIVERY AT SPEED

28-29 OCT, 2019
GDANSK, POLAND

BOOK YOUR TICKET NOW!

REJESTRACJA.AADAYS.PL

AGILE & AUTOMATION DAYS | CONFERENCE & WORKSHOPS | 28-29 OCTOBER 2019 GDANSK, POLAND

```
for m2 in range(2 ** 22, 2 ** 25):
    m2_inv = modinv(m2, N)
    left = ct * pow(m2_inv, e, N)

    if left in right_results:
        m1 = right_results[left]
        print "Found M: ", m1 * m2
```

I to w zasadzie tyle – zredukowaliśmy ilość operacji potrzebnych do rozwiązania zadania z wielkości rzędu 2^{46} do $2*2^{25}=2^{26}$. Taka taktyka liczenia równania „z dwóch stron jednocześnie" nazywa się właśnie *Meet In The Middle*, która jest dość popularnym atakiem w kryptografii i, jak widać, potrafi znacząco przyspieszyć atak[10].

To wszystko oczywiście przy założeniu, że `M` spełnia taki warunek, jaki założyliśmy na początku (jest iloczynem dwóch dużych liczb). Okazuje się, że w praktyce jest na to całkiem spora szansa – i faktycznie, zachodzi taka sytuacja także w przypadku `M` podanego w zadaniu.

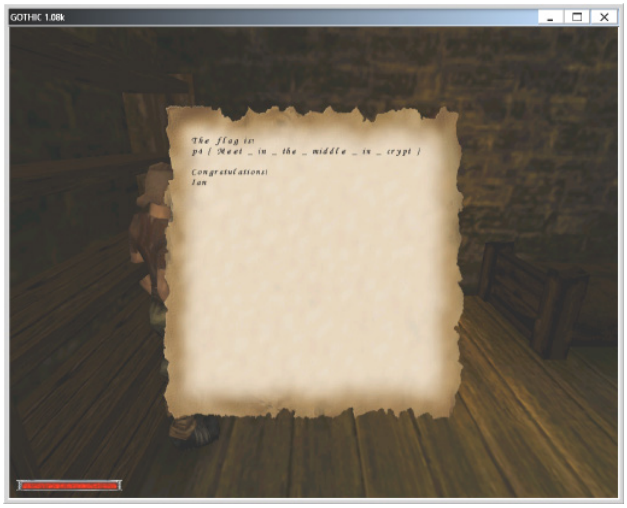
Na zakończenie warto dodać, że mimo wszystko jest to pewna podatność w RSA (atakujący nie powinien być w stanie tak łatwo odzyskać oryginalnej wiadomości, nawet jeśli spełnia jakieś szczególne warunki). Jest to znany problem z tzw. „książkowym RSA" (ang. *textbook rsa*), dlatego w praktyce zawsze stosuje się *padding* na szyfrowanej wiadomości.[11]

Teraz trzeba domyślić się, że liczba `M` pomoże nam otworzyć skrzynię ze skarbem (sugeruje to np. taka sama ilość bitów `M` jak długość kodu do skrzyni). W tym celu musimy ją zamienić na liczbę w systemie dwójkowym, a następnie przepisać w grze (to ta trudniejsza część…). Każde 0 w liczbie reprezentuje ruch wytrychem „w lewo", a każde 1 ruch „w prawo" (Rysunek 3.3).

Rysunek 3.3. Otwieranie skrzyni

Na koniec wystarczy wyciągnąć ze skrzyni księgę, na której zapisana jest flaga (Rysunek 3.4). Brzmi ona:

```
p4{Meet_in_the_middle_in_crypt}.
```



Rysunek 3.4. Flaga

ZAKOŃCZENIE

Zadania *Gothic* od strony technicznej miały dość standardowy charakter, ale wyróżniały się ciekawym podejściem do tematu i doskonałą oprawą audio-wizualną. Mimo drobnych problemów technicznych (na przykład z uruchomieniem gry w środowisku Wine u niektórych) zadanie zebrało pochwały chyba od wszystkich graczy, którzy się z nim zmierzyli.

Jeśli ktoś chce spróbować swoich sił, te (oraz inne) zadania ciągle są dostępne pod adresem https://confidence2019finals.p4.team. Zachęcamy do zapoznania się z nimi zwłaszcza tych, którzy nie mieli możliwości pojawić się osobiście na konferencji.

Jarosław Jedynak

Rozwiązanie zadań *Gothic RE* oraz *Gothic Crypto* zostało nadesłane przez p4, polski zespół CTF-owy, który jest królem CTFów jak lew jest królem dżungli (https://ctftime.org/team/5152).

» https://ctftime.org/team/3329
» https://ctftime.org/ctf/103
» https://www.instytutpwn.pl/konferencja/pwning/

10. Szczególnie że ten atak MITM również można zrównoleglić. Przykładowy wzorcowy kod autorstwa zawodnika Shalom z zespołu p4 (autora zadania) można znaleźć pod adresem https://store.tailcall.net/gothic_crypto.py.

11. Więcej na ten temat można poczytać np. w opracowaniu *Why Textbook ElGamal and RSA Encryption Are Insecure* (Dan Boneh, Antoine Joux, Phong Q. Nguyen), dostępnej publicznie pod adresem https://link.springer.com/content/pdf/10.1007/3-540-44448-3_3.pdf.