MAKALAH TUGAS AKHIR

PEMROGRAMAN APLIKASI PROTEKSI FILE *EXECUTABLE* BERBASISKAN PE *(PORTABLE EXECUTABLE)* DENGAN MENGGUNAKAN KOMPILER MASM 32 (*MICROSOFT MACRO ASSEMBLER 32*)

Oleh :
SALMON GUNAWAN
L2F098654

Abstrak

Dewasa ini banyak sekali program-program aplikasi yang beredar di kalangan pengguna komputer, baik didistribusikan melalui media *compact disk* (CD) maupun dapat diambil secara langsung melalui jaringan internet dengan terlebih dahulu men-*dowload*-nya. Program-program aplikasi yang beredar tersebut sebagian besar didistribusikan sebagai produk yang diperdagangkan (*shareware*), serta sebagian kecil lagi diberikan secara gratis (*freeware*). Program aplikasi yang diperdagangkan tersebut pada umumnya memiliki teknik proteksi yang sangat lemah terhadap teknik–teknik *debugging, disassambing* serta *patching.* Teknik-teknik tersebut biasanya didukung oleh berbagai program *debugger* dan *disassambler* seperti SoftIce, URSoft W32Dasm serta program lainnya yang banyak beredaran di internet.

Sehingga tujuan dari Tugas Akhir ini adalah untuk menghasilkan program yang memberikan berbagai sistim proteksi bagi file *executable* dari kegiatan *debugging, disassambing* serta *patching* (pengubahan kode) tanpa perlu mengkompilasi ulang program *executable* tersebut. Teknik yang digunakan adalah dengan menyisipkan/menginjeksikan sekumpulan rutin fungsi ke dalam badan program yang memiliki format PE (*Portable Executable),* yaitu suatu bentuk standarisasi format executable yang berjalan di sistim operasi Microsoft Windows 95/98 dan Windows NT/2000.

File PE yang telah diinjeksi akan bertambah besar ukurannya sebesar 2 sampai 3 KByte, disertai dengan pengenkripsian semua data (*Raw Section Data*) pada direktori seksi (*Section Directory*). Akan tetapi kecepatan proses eksekusi dari file PE hasil injeksi tidak mengalami perubahan yang signifikan oleh karena pertambahan ukuran memori virtual hanya sebesar 2 KByte. Ukuran memori virtual sebesar 2 Kbyte tersebut diperuntukkan bagi eksekusi program *loader* beserta rutin-rutin proteksi yang diinjeksikan.

I PENDAHULUAN

1.1 Latar Belakang

Dewasa ini banyak sekali program-program aplikasi yang beredar di kalangan pengguna komputer, baik didistribusikan melalui media *compact disk* (CD) maupun dapat diambil secara langsung melalui jaringan internet dengan terlebih dahulu men-*dowload*-nya. Sebagian besar program aplikasi tersebut berjalan di sistim operasi Windows. Hal ini didukung oleh sebagian besar pengguna komputer di seluruh dunia yang menggunakan sistim operasi tersebut.

Program-program aplikasi yang beredar tersebut sebagian besar didistribusikan sebagai produk yang diperdagangkan (*shareware*), serta sebagian lagi diberikan secara gratis (*freeware*). Produk – produk *shareware* tersebut pada umumnya memiliki pola-pola proteksi berupa :

1. Adanya mekanisme form registrasi (pendaftaran) dari nama user serta nomor serial atau password (*serial number*), dimana hal tersebut diperoleh setelah pengguna membayar lisensinya melalui media internet atau *electronic data exchange* seperti kartu kredit.
2. Adanya fasilitas pembatasan jangka waktu pemakaian pada program aplikasi yang bersifat uji coba (*demo version*). Sehingga apabila pengguna telah melewati batas waktu penggunaan, maka perangkat lunak (*shareware*) tersebut tidak dapat berfungsi lagi. Program *shareware* akan berfungsi kembali setelah pengguna membeli lisensi dari pembuat perangkat lunak tersebut.
3. Adanya mekanisme pembatasan beberapa fungsi utama program. Sehingga agar program dapat berfungsi secara penuh, pengguna perlu meregistrasinya terlebih dahulu.

Pada kenyataannya berbagai sistim proteksi diatas yang diberikan oleh berbagai program *shareware* tersebut dapat dengan mudah ditembus oleh *cracker*. Umumnya seorang *cracker* menggunakan teknik *debugging* dan *disassambling* untuk menganalisa berbagai sistim proteksi tersebut. Penganalisaan program dengan menggunakan teknik *debugging* dan *disassambling* pada umumnya dalam bentuk bahasa *assembly* 32 bit atau lebih dikenal dengan bahasa pemrograman *Win32Asm*. Sayangnya, sebagian besar pengembang perangkat lunak bekerja pada lingkungan pemrograman tingkat tinggi (*High Level Language*) serta tidak memiliki pengetahuan yang cukup mengenai sistim proteksi file PE terhadap segala teknik rekayasa balik (*debugging, disassambling, code editing*) yang bekerja dalam lingkungan bahasa tingkat rendah (*assembly*).

Oleh sebab itu perlu adanya mekanisme sistem proteksi yang handal yang dapat mencegah teknik-teknik rekayasa balik yang digunakan oleh *cracker*, serta dapat memberikan kemudahan bagi para pengembang perangkat lunak tanpa perlu mempelajari segala teknik rekayasa balik yang dilakukan oleh cracker. Sehingga para pengembang perangkat lunak (*shareware*) dapat lebih memusatkan pada kualitas produknya (*shareware*), tanpa perlu dipusingkan dengan sistim proteksi file *executable* yang harus diberikan.

Kemudahan inilah yang diberikan oleh hasil dari Tugas Akhir ini, yaitu dengan memasukkan atau menginjeksikan secara langsung sejumlah fungsi proteksi terhadap berbagai teknik rekayasa balik, pada program *executable* yang telah dikompilasi, tanpa perlu melakukan proses kompilasi ulang program *executable* tersebut.

Berbagai penelitian tentang sistim proteksi file PE telah banyak dilakukan oleh sejumlah pengembang perangkat lunak antara lain, teknik *Dongle*[13] yang dikembangkan oleh Rainbow Technologies (USA) dengan nama produk Sentinel dan Aladin Software (Israel) dengan nama produk HASP, teknik *MeltIce*[13] oleh David Eriksson, teknik *FLEXcrypt*[13] yang dibuat oleh Globetrotter dengan algoritma XOR berputar, serta berbagai teknik kompresi file PE (EXE *Packer*[13]) yang ditunjukkan pada program-program seperti Shrinker, WWPack32, NeoLite, ASPack.

1.2 Tujuan Penulisan

Tujuan pembuatan Tugas Akhir ini adalah sebagai berikut:

1. Menjelaskan struktur PE (*Portable Executable)* yang berhubungan dengan teknik penyisipan kode pada file *executable* dengan menggunakan bahasa pemrograman MASM 32 (*Microsoft Macro Assembler 32*).
2. Mengaplikasikan beberapa metoda proteksi file PE, yaitu diantaranya anti *debugger* SoftIce (metoda *MeltIce*, *Bounds Checker*), anti *disassembler* W32Dasm (API *Redirection*, menghapus *Image Import Descriptors*), anti pengubahan kode (teknik ceksum / *Cyclic Redudancy Code check*), yang masing-masing dapat berguna dalam pencegahan teknik rekayasa balik (*reverse engineering*) yang biasa dilakukan *cracker*.

1.3 Pembatasan Masalah

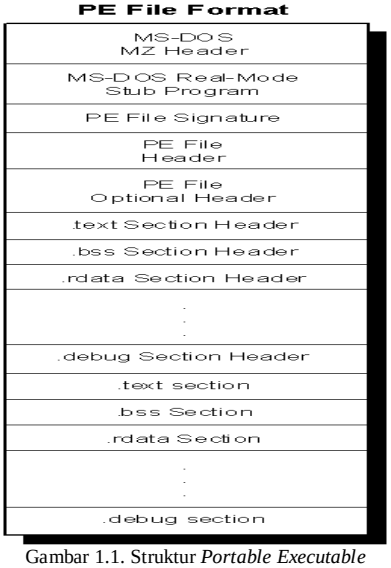
Dalam Tugas Akhir ini penulis akan membuat batasan permasalahan agar tidak menyimpang dari pokok pembahasan yang sebenarnya. Hal-hal yang dibuat dan dibahas dalam tugas akhir ini adalah sebagai berikut :

1. Pembahasan struktur *Portable Executable* (PE) sebagai format binary dasar bagi program aplikasi (baik berupa *executable / EXE* maupun *library* / DLL) yang berjalan di atas sistim operasi MS Windows, khususnya ber-platform Win32.
2. Pembuatan program dalam bahasa *Assembly* 32 bit dengan menggunakan kompiler MASM 32 (*Microsoft Macro Assembler 32*) versi 7.
3. Pembahasan teknik – teknik proteksi file *executable* dari segala bentuk kegiatan *debugging*, *disassambling* serta *patching* (pengubahan kode program).
4. Pengujian program dilakukan pada program-program *debugger* atau *disassambler* yang banyak digunakan oleh *cracker*, yaitu SoftIce versi 4.05 serta URSoft W32Dasm versi 8.93.

II LANDASAN TEORI

Portable Executable (PE)

Portable Executable (PE) adalah format binari dasar bagi program aplikasi (baik berupa *executable* / .EXE maupun *library* / .DLL) yang berjalan di atas sistim operasi Microsoft Windows 95/98 serta Windows NT/2000, khususnya yang ber-platform Win32 serta didalamnya memiliki format COFF (*Common Object File Format*). COFF adalah bentuk dasar dari file obyek (OBJ *file / object file*) dan *executable* yang digunakan oleh beberapa sistim operasi seperti Microsoft, UNIX, VAX serta VMS. Format PE tersebut merupakan disain dasar dari produk Microsoft Corp. serta didukung oleh berbagai perusahaan perangkat lunak lainnya yang tergabung dalam suatu komite bernama TIS (*Tool Interface Standard*) Committee, yang beranggotakan Microsoft, Intel, Borland, Watcom, IBM, serta perusahaan lainnya.



Gambar 1.1. Struktur *Portable Executable*

Format PE tersebut pertama kali diperkenalkan oleh Microsoft pada sistim operasi Windows Nt™ versi 3.1. Kata *Portable* memiliki arti bahwa format PE tersebut memiliki implementasi yang sama untuk bermacam-macam prosesor seperti DEC Alpha AXP, INTEL x86,

MIPS, Motorola 68000 *series*, IBM Power PC. Dengan kata lain format PE dibuat oleh Microsoft dengan tujuan utama yaitu multiplatform. Berikut ini akan dijelaskan secara lengkap struktur *Portable Executable* (PE).

2.1.1 MS-DOS *Header* (IMAGE_DOS_HEADER)

Konsep dasar MS-DOS telah lama digunakan pada program-program aplikasi Windows 16 bit. Pada pemrograman Win32, MS-DOS *header* hanya digunakan sebagai validasi bahwa program aplikasi tersebut kompatibel dengan MS-DOS versi 2.0. Walaupun demikian program tersebut tidak dapat dijalankan dalam mode MS-DOS dan akan menampilkan pesan "*This program must be run under Win32*" atau "*This program cannot be run in DOS mode*". MS-DOS *header* merupakan suatu struktur dari IMAGE_DOS_HEADER. Berikut ini adalah bentuk struktur dari IMAGE_DOS_HEADER dalam format bahasa C :

```
typedef struct _IMAGE_DOS_HEADER {
    USHORT e_magic;         // nilai
identitas
    USHORT e_cblp;          // jumlah BYTE
pada page terakhir file
    USHORT e_cp;            // jumlah page
pada file
    USHORT e_crlc;          // relokasi
    USHORT e_cparhdr;       // ukuran header
pada paragraf
    USHORT e_minalloc;      // nilai minimum
ekstra paragraf
    USHORT e_maxalloc;      // nilai
maksimum ekstra paragraf
    USHORT e_ss;            // inisialisasi
(relatif) nilai SS (Stack Segment)
    USHORT e_sp;            // inisialisasi
nilai SP (Stack Pointer)
    USHORT e_csum;          // checksum
    USHORT e_ip;            // inisialisasi
nilai IP (Index Pointer)
    USHORT e_cs;            // inisialisasi
(relatif) nilai CS (Code Pointer)
    USHORT e_lfarlc;        // alamat file
dari tabel relokasi
    USHORT e_ovno;          // nilai overlay
    USHORT e_res[4];        // nilai
reserved 1 WORD
    USHORT e_oemid;         // identitas OEM
    USHORT e_oeminfo;       // informasi OEM
    USHORT e_res2[10];      // nilai
reserved 1 WORD
    LONG   e_lfanew;        // alamat file
dari header executable baru
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Baris pertama dari struktur MS-DOS *header* diatas diawali oleh suatu nilai identitas (*e_magic*) yang selalu bernilai heksadesimal 0x54AD atau dalam bentuk ASCII (*American Standard Code for Information Interchange*) yaitu "MZ" (berukuran 2 BYTE). Baris terakhir dari struktur MS-DOS *header* (*e_lfanew*) adalah suatu identitas bagi validasi program *executable* yang berbasiskan PE dengan nilai berukuran 32 bit (1 DWORD) serta lebih dikenal sebagai IMAGE_NT_SIGNATURE. Identitas tersebut dalam bilangan heksadesimal bernilai 0x00004550 atau dalam bentuk ASCII yaitu string "PE..".

2.1.2 File *Header* (IMAGE_FILE_HEADER)

File *Header* merupakan bentuk struktur dari IMAGE_FILE_HEADER yang berada tepat setelah IMAGE_NT_SIGNATURE teridentifikasi. Berikut ini adalah struktur dari IMAGE_FILE_HEADER :

```
typedef struct _IMAGE_FILE_HEADER {
        USHORT  Machine;
        USHORT  NumberOfSections;
        ULONG   TimeDateStamp;
        ULONG   PointerToSymbolTable;
        ULONG   NumberOfSymbols;
        USHORT  SizeOfOptionalHeader;
        USHORT  Characteristics;
}IMAGE_FILE_HEADER,*PIMAGE_FILE_HEADER;
```

Keterangan dari masing-masing adalah sebagai berikut :

a. *Machine.*
- Berupa bilangan heksadesimal berukuran 16 bit (1 WORD) yang menunjukkan jenis prosesor yang dipakai oleh program *executable* tersebut. Angka ini berhubungan dengan kompatibilitas program, yaitu di lingkungan mana program tersebut dapat dijalankan.

b. *NumberOfSections.*
- Berupa bilangan heksadesimal berukuran 16 bit (1 WORD) yang merupakan jumlah daripada *Section Header* yang terdapat pada file PE. Jumlah *Section Header* juga menyatakan jumlah *Section Directory* yang merupakan pusat data sesungguhnya dari sebuah file PE.

c. *TimeDateStamp.*
- Bilangan heksadesimal berukuran 32 bit (1 DWORD) yang menyatakan waktu dan tanggal executable tersebut dihasilkan.

d. *PointerToSymbolTable.*
- Bilangan heksadesimal berukuran 32 bit (1 DWORD) yang menyimpan informasi tentang *debugging* proses.

e. *NumberOfSymbols.*
- Sama seperti PointerToSymbolTable, yaitu berupa bilangan heksadesimal berukuran 32 bit (d1 WORD) yang menyimpan informasi tentang *debugging.*

f. *SizeOfOptionalHeader.*
- Bilangan heksadesimal berukuran 16 bit (1 WORD) yang merupakan ukuran dari IMAGE_OPTIONAL_HEADER.
- Dapat digunakan untuk mengecek apakah file *executable* sudah memenuhi struktur PE yang benar.

g. *Characteristics.*
- Bilangan heksadesimal berukuran 16 bit (1 WORD) yang terdiri dari serangkaian flag serta valid apabila digunakan pada *object file* dan *library.*

2.1.3 RVA (Relative Virtual Address)

RVA (*Relative Virtual Address*) adalah suatu nilai alamat memori virtual dimana nilai tersebut belum ditambahkan dengan nilai alamat *Image Base* jika ingin di-*load* di memori. *Image Base* adalah alamat awal suatu image PE yang akan dimuat ke dalam memori. Dengan kata lain alamat linear / VA (*Virtual Address*) adalah alamat RVA ditambahkan dengan alamat *Image Base*.

Sebagai contoh alamat image PE yang dimuatkan ke dalam ruang alamat memori bernilai 401000h, serta alamat Image Base sebesar 400000h. Maka alamat RVA dari PE tersebut adalah sebesar 1000h.

2.1.4 Optional Header (IMAGE_OPTIONAL_HEADER)

Struktur berikutnya dalam format PE adalah *Optional Header* yang didefinisikan sebagai IMAGE_OPTIONAL_HEADER serta berukuran 224 BYTE. Image tersebut diatas berisi informasi penting mengenai rancangan logika dari sebuah file PE yang sangat dibutuhkan oleh loader.

Struktur *Optional Header* memiliki 31 bagian (*field*). Akan tetapi tidak semua bagian dari struktur *Optional Header* yang dipergunakan. Hanya beberapa bagian saja yang sangat penting sedangkan lainnya tidak begitu dibutuhkan. Berikut ini adalah struktur dari IMAGE_OPTIONAL_HEADER :

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //
    USHORT  Magic;
    UCHAR   MajorLinkerVersion;
    UCHAR   MinorLinkerVersion;
    ULONG   SizeOfCode;
    ULONG   SizeOfInitializedData;
    ULONG   SizeOfUninitializedData;
    ULONG   AddressOfEntryPoint;
    ULONG   BaseOfCode;
    ULONG   BaseOfData;
    //
    // NT additional fields.
    //
    ULONG   ImageBase;
    ULONG   SectionAlignment;
    ULONG   FileAlignment;
    USHORT  MajorOperatingSystemVersion;
    USHORT  MinorOperatingSystemVersion;
    USHORT  MajorImageVersion;
    USHORT  MinorImageVersion;
    USHORT  MajorSubsystemVersion;
    USHORT  MinorSubsystemVersion;
    ULONG   Win32VersionValue;
    ULONG   SizeOfImage;
    ULONG   SizeOfHeaders;
    ULONG   CheckSum;
    USHORT  Subsystem;
    USHORT  DllCharacteristics;
    ULONG   SizeOfStackReserve;
    ULONG   SizeOfStackCommit;
    ULONG   SizeOfHeapReserve;
    ULONG   SizeOfHeapCommit;
    ULONG   LoaderFlags;
    ULONG   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY
DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTR
IES];
} IMAGE_OPTIONAL_HEADER,
*PIMAGE_OPTIONAL_HEADER;
```

Tampak bahwa pada struktur diatas terdapat dua sub kelompok field, yaitu kelompok *Standard field* dan kelompok *NT additional field. Standard field* biasanya merupakan file obyek (*object file*) yang berguna untuk memenuhi struktur COFF (*Common Object File Format*), serta biasanya banyak digunakan oleh file-file *executable* UNIX. *Standard field* berisi informasi yang sangat penting bagi file PE, yaitu bagaimana file PE tersebut di-*load* serta berjalan di memori. Sedangkan *NT additional field* berisi informasi yang dibutuhkan oleh *linker* (penggabung kode) dan *loader* (pembangkit kode) pada Windows NT.

2.1.5 Direktori Seksi (*Section Directory*)

Direktori seksi adalah bagian dari PE yang merupakan sumber data program sesungguhnya, serta berada setelah *Optional Header*. Direktori seksi terdiri dari dua bagian utama, yaitu sebuah deskripsi seksi (IMAGE_SECTION_HEADER) serta bagian data (*Raw Section Data*). Deskripsi seksi (IMAGE_SECTION_HEADER) untuk berikutnya dikenal sebagai *Section Header*.

Seperti yang sudah dijelaskan sebelumnya pada *File Header* bahwa jumlah daripada keseluruhan seksi disimpan dalam field *NumberOfSections* yang mengacu pada jumlah daripada *Section Header*. Struktur bagian daripada *Section Header* adalah sebagai berikut :

```
#define IMAGE_SIZEOF_SHORT_NAME    8
typedef struct _IMAGE_SECTION_HEADER {
    UCHAR
Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            ULONG   PhysicalAddress;
            ULONG   VirtualSize;
    } Misc;
    ULONG   VirtualAddress;
    ULONG   SizeOfRawData;
    ULONG   PointerToRawData;
    ULONG   PointerToRelocations;
    ULONG   PointerToLinenumbers;
    USHORT  NumberOfRelocations;
    USHORT  NumberOfLinenumbers;
    ULONG   Characteristics;
}                   IMAGE_SECTION_HEADER,
*PIMAGE_SECTION_HEADER;
```

Penjelasan untuk masing-masing bagian adalah sebagai berikut :

1. *Name.*

- Merupakan larik dari IMAGE_SIZEOF_SHORT_NAME yang menghasilkan sebuah nama seksi (*Section Name*) dalam format string ASCII. String tersebut berukuran 8 Byte serta tidak diakhiri oleh 00h (*zero / null terminated*). Apabila lebih dari 8 Karakter maka nama seksi tersebut akan terpotong. Format *Section Name* pada umumnya diawali dengan tanda titik seperti ".idata, ".text", ".data", ".bss". Akan tetapi tidak semua nama seksi diawali oleh tanda titik, contoh "CODE", "IAT".

2. *PhysicalAddress, VirtualSize.*

- *PhysicalAddress* dan *VirtualSize* adalah merupakan field gabungan serta merupakan *object file*. Sebagai *object file* keduanya merupakan alamat dan ukuran dari isi sebuah relokasi file *executable*. Akan tetapi pada prakteknya kedua field gabungan tersebut tidak berguna sama sekali.

3. *VirtualAddress.*
- Menyimpan informasi dari RVA (alamat relatif) *Section Data* pada saat akan dimuatkan ke dalam memori (RAM).
- Informasi ini berukuran 32 bit (1 DWORD).

4. SizeOfRawData.
- Menyimpan informasi ukuran dari *Section Data* yang merupakan nilai relatif terhadap *FileAlignment*.
- Informasi ini berukuran 32 bit (1 DWORD).

5. PointerToRawData.
- Merupakan bagian yang sangat penting, yaitu menunjuk pada sebuah offset dari lokasi *Section Body* pada file PE. *Section Body* merupakan tempat data sesungguhnya berada (hasil kompilasi).
- Informasi ini berukuran 32 bit (1 DWORD).

6. PointerToRelocations.
- Sebagai OBJ (*Object File)*, *PointerToRelocations* merupakan pointer pada lokasi awal input relokasi jika terdapat pada *section*. Sedangkan pada file PE (*executable*) informasi tersebut tidak dibutuhkan serta nilainya selalu 0.
- Informasi ini berukuran 32 bit (1 DWORD).

7. PointerToLinenumbers.
- Merupakan pointer yang mengacu pada larik jumlah baris (*LineNumber*) pada file *executable* yang hanya dibutuhkan sebagai *object file*. *PointerToLinenumbers* biasanya digunakan sebagai informasi tambahan format *debug*.
- Informasi ini berukuran 32 bit (1 DWORD).

8. NumberOfRelocations.
- Merupakan jumlah relokasi pada tabel relokasi untuk masing-masing seksi (section) yang ditunjuk oleh *PointerToRelocations.*
- Informasi ini berukuran 16 bit (1 WORD).

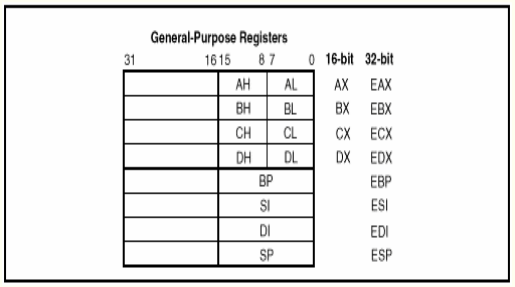
9. NumberOfLinenumbers.
- Merupakan jumlah *LineNumber* pada tabel *LineNumber* untuk masing-masing seksi (*section*) yang ditunjuk oleh *PointerToLinenumbers*.
- Informasi ini berukuran 16 bit (1 WORD).

10. Characteristics.
- Field berukuran 32 bit (DWORD) ini berisi flag-flag informasi mengenai atribut / karakteristik *section*, yaitu apakah sebuah *Data Section* berupa kode (*code / executable*) , data *readable* (hanya dapat dibaca) serta data *writeable* (data dapat diubah).

Setelah *Section Header* maka struktur berikutnya adalah bagian data utama (*Raw Section Data*), yaitu tempat data file *executable* itu sesungguhnya. Pada bagian ini berisi segala data hasil kompilasi dan penyatuan oleh *compiler* dan *linker*. Sebuah file PE pada Windows NT memiliki 9 definisi nama *section*, yaitu "text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, dan .debug". Pada kenyataanya tidak semua aplikasi memiliki kesembilan nama *section* tersebut. Masing-masing nama *section* tersebut pada umumnya mewakili fungsi data yang berbeda-beda.

2.2 Pemrograman Bahasa Assembler 32 bit (Win32Asm) dengan MASM 32

Sejarah pemrograman Win32 dimulai sejak munculnya prosesor Intel 80386. Generasi awal dari lingkungan Win32 adalah munculnya sistim operasi Windows 3.0. Sedangkan munculnya konsep PE (PortableExecutable) yang mendukung pemrograman Win32 adalah saat dikeluarkannya Windows Nt versi 3.1 oleh *Microsoft Corp*. Pemrograman Win32 memiliki kemampuan mengakses ruang alamat memori sebesar 4 GB (Giga Byte). Tetapi tidak berarti bahwa semua program berbasis Win32 menggunakan keseluruhan dari ruang memori fisik tersebut. Hal ini berbeda sekali dengan lingkungan Win16 (sistim operasi DOS), dimana program-program Win16 hanya dapat mengakses ruang alamat memori DOS sebesar 640 KB (Kilo Byte). Selain itu pemrograman Win16 juga dibatasi oleh adanya pemakaian segmen-segmen memori yang hanya berukuran 64 KB serta ukuran register memori sebesar 16 bit (AX,BX,CX). Sedangkan pada lingkungan Windows, kebutuhan akan segmen-segmen memori tersebut tidak efektif lagi. Karena pada lingkungan Win32 (Windows) hanya memiliki satu model memori, yang bernama *flat memory model*. Selain itu pada lingkungan pemrograman Win32 memiliki kapasitas register memori sebesar 32 bit (*Extended* AX/EAX,EBX,ECX). Berikut ini adalah gambar ruang register memori pada lingkungan Win32.



Gambar 2.1. Ruang Register Memori pada Lingkungan Win32

Pemrograman Win32Asm adalah suatu bentuk bahasa pemrograman di lingkungan *assembly* yang memanfaatkan segala kelebihan aplikasi Win32 atau yang lebih dikenal sebagai Windows 32 bit. Kelebihan-kelebihan yang dimiliki bahasa pemrograman Win32Asm

dibandingkan dengan bahasa tingkat tinggi seperti bahasa C adalah sebagai berikut :

- Hasil kompilasi program memiliki ukuran yang sangat kecil.
- Program yang dihasilkan relatif sangat cepat saat dieksekusi.
- Sangat mudah dalam berinteraksi dengan perangkat keras (*hardware*) sehingga sangat cocok dalam pembuatan aplikasi *driver / library*.

MASM32 (*Microsoft Macro Assembler 32*) merupakan salah satu kompiler bahasa *assembly* 32 bit yang diproduksi oleh Microsoft. MASM32 memiliki beberapa kelebihan yang tidak dimiliki oleh kompiler lain seperti TASM (*Borland Turbo Assembler*) serta NASM, yaitu diantaranya :

- MASM32 memiliki kemampuan bahasa tingkat tinggi ,yaitu adanya sintaks perbandingan dan perulangan (looping) seperti ".IF, .ELSE, .ELSEIF, .ENDIF, .REPEAT, .UNTIL, .WHILE, .ENDW, .BREAK, .CONTINUE ".

Contoh :

A) .IF EAX == 1
   XOR EAX,EAX
   MOV EAX,ECX
   .ENDIF

B) .WHILE EAX!=0
   INC ECX
   DEC EAX
   .ENDW

- Adanya suatu fungsi pemanggil bernama INVOKE yang mempermudah dalam pemanggilan fungsi / prosedur.

❖ Sintaksnya adalah : **INVOKE expression [,arguments]**

Contoh :

A.) INVOKE procedure, parameter1, parameter2, parameter3

Identik dengan bentuk **:**

PUSH parameter3
PUSH parameter3
PUSH parameter3
CALL procedure

B.) INVOKE MessageBoxA, ADDR szMsg, ADDR szTitle, MB_OK

- Adanya MACRO yang merupakan keunggulan utama dari MASM32. MACRO tersebut sangat berguna dalam kecepatan oleh karena mendukung adanya pemrosesan awal (*pre processing*) dari suatu atau beberapa kode sebelum *assembler* menganalisa dan menterjemahkannya.

❖ Sintaksnya adalah :

name MACRO [parameter[:tag]]
[,parameter[:tag]]...]
[LOCAL varlist]
statements
[EXITM [textitem]]
ENDM

Contoh :

A.) PUPO MACRO pSrc, pDest
PUSH pSrc
POP pDest
ENDM

B.) szText MACRO Name, Text:VARARG
LOCAL lbl
jmp lbl
Name db Text,0
lbl:
ENDM

Berikut ini adalah kerangka struktur kode pemrograman MASM32 :

```
.386
.MODEL Flat, STDCALL
option casemap:none
.DATA
    <Data yang sudah diinisialisasi>
    ......
.DATA?
   < Data yang belum diinisialisasi
nilainya>
   ......
.CONST
   <nilai konstanta>
   ......
.CODE
   <label>
    <rutin kode>
   .....
    end <label>
```

Penjelasan untuk masing-masing baris sintaks adalah sebagai berikut :

❖ **.386**

- Merupakan kode direktif assembler yang menggunakan kompatibilitas serta optimasi dari *instruction set* tipe prosesor Intel 80386. Beberapa kode direktif yang dapat digunakan adalah "**.386/.386p, .486/.486p, .586/586p**" serta generasi prosesor Intel diatasnya, seperti Pentium. Kode **p** di belakang tipe prosesor menunjukkan bahwa program yang dihasilkan memiliki keistimewaan instruksi (*priveleged instruction*) serta biasanya digunakan pada program driver (**VxD** – *Virtual device Driver*).

❖ **.MODEL Flat, STDCALL**

- .**Model Flat** adalah kode direktif assembler yang menunjukkan model memori yang digunakan pada lingkungan Windows (Win32).
- **STDCALL** adalah standarisasi pengiriman parameter-parameter fungsi atau prosedur, yaitu dikirim dari kanan ke kiri (format bahasa C) atau sebaliknya (format bahasa Pascal).

❖ **Option casemap:none**

- Berfungsi agar label-label bersifat *case sensitive* (adanya perbedaan antara penggunaan huruf besar dan kecil).

❖ **.DATA**

- Berisi data-data yang telah di inisialisasi (*initialized data*).

❖ **.DATA?**

- Berisi data-data yang belum terinisialisasi (***un****initialized data*).

❖ **.CONST**

- Berisi deklarasi konstanta yang digunakan oleh program.

❖ **.CODE**

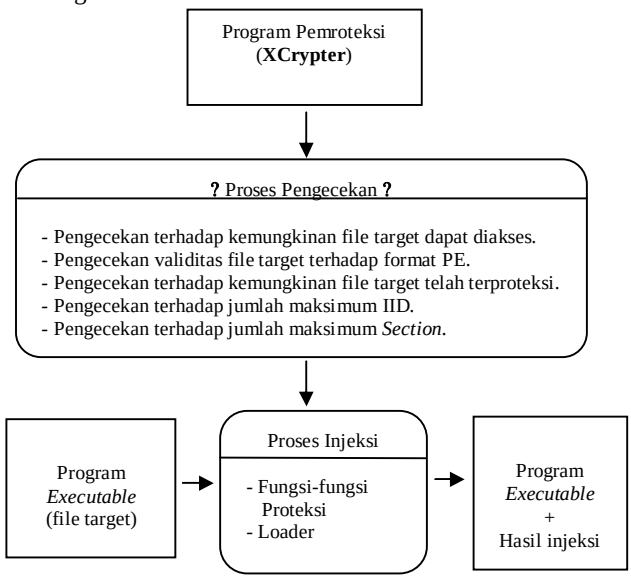
- Berisi kode-kode baik prosedur maupun fungsi yang berjalan pada program.

2.3 Teknik – Teknik Rekayasa Balik (*Reverse Engineering*)

Banyak sekali teknik-teknik yang digunakan oleh seorang *cracker* untuk melakukan usaha / teknik rekayasa balik (*Reverse Engineering*) pada berbagai program *executable*. Semakin berpengalaman seorang *cracker* dalam menganalisa dan merekayasa balik program-program terproteksi, maka semakin banyak pula teknik-teknik yang digunakannya. Seorang *cracker* /*hacker* dapat juga dikatakan sebagai seorang sistim analis, yaitu satu tingkat di atas programer. Pada umumnya teknik-teknik yang digunakan tersebut tergantung pada jenis-jenis proteksi yang dipakai pada program-program komersil tersebut (*shareware, demo version*). Akan tetapi metode umum dari teknik-teknik yang digunakan oleh para cracker untuk melakukan aksinya adalah dengan melakukan *debugging*, *disassembling* serta *patching*.

III IMPLEMENTASI PROGRAM

Secara garis besar program pemroteksi yang dibuat memberikan pola-pola sistim proteksi pada berbagai program *executable* (terutama yang bersifat *shareware*) dengan cara menyuntikkan sejumlah kode atau fungsi dalam bahasa *assembly* ke dalamnya. Sebagai gambaran umum mengenai sistim secara keseluruhan dapat dilihat pada gambar 3.1.



Gambar 3.1. Gambaran Umum Sistim

Dari ilustrasi di atas tampak jelas bahwa program hasil injeksi tidak berubah fungsi sama sekali, hanya saja mendapatkan tambahan sejumlah kode program yang berfungsi sebagai protektor. Selain itu dari gambar tersebut tampak jelas bahwa program *executable* (file target) tidak perlu mengalami proses kompilasi ulang.

3.1 Proses-Proses Pengecekan

Pada saat file target akan diinjeksi, terlebih dahulu program pemroteksi melakukan beberapa algoritma pengecekan terhadap file target, yaitu :

3.1.1 Pengecekan terhadap Kemungkinan File Target dapat Diakses

Pengecekan ini dilakukan untuk memastikan bahwa file target (*executable*) dapat diakses atau tidak. Pada proses pengecekan ini harus memenuhi tiga tahapan validasi, yaitu:

1. File target harus benar-benar ada.
2. File target harus memiliki ukuran yang lebih besar dari 0 byte (tidak kosong).
3. Kebutuhan memori file target harus memenuhi kapasitas memori perangkat keras yang digunakan.

3.1.2 Pengecekan Validitas File Target terhadap Format PE (*Portable Executable*)

Pengecekan ini dilakukan untuk memastikan bahwa file target merupakan program *executable* yang memenuhi standar format PE.

3.1.3 Pengecekan terhadap Kemungkinan File Targe telah Terproteksi

Pengecekan ini dilakukan dengan tujuan agar file target tidak mengalami proses penginjeksian untuk kedua kalinya. Dengan kata lain file target yang sudah terproteksi tidak dapat diinjeksi ulang dengan program pemroteksi ini.

3.1.4 Pengecekan terhadap Jumlah Maksimum *Image Import Descriptor*

Pengecekan ini dilakukan terhadap ukuran *Image Import Descriptor* (IID) yang menyimpan semua informasi beserta alamat relatif dari segala impor fungsi (API – *Aplication Programming Interface*). Nilai maksimum dari ukuran IID dibatasi sebanyak 30 buah. Sedangkan format sebuah larik (array) IID terdiri dari 5 DWORD (5 x 32 Bit).

3.1.5 Pengecekan terhadap Jumlah Maksimum *Section*

Tahap pengecekan ini dilakukan oleh karena fungsi-fungsi proteksi serta program *loader* yang diinjeksikan, diproses dengan cara membuat sebuah *section* yang baru pada direktori seksi (*Section Directory*). Sehingga sebelum menambah sebuah *section*, harus membandingkan terlebih

dahulu jumlah direktori seksi setelah ditambah dengan jumlah maksimum direktori seksi yang diperbolehkan (sebesar 20 buah).

3.2 Proses Injeksi

Proses injeksi dibagi menjadi dua bagian utama, yaitu terdiri dari:

❖ Fungsi-fungsi Kode Proteksi .

- Yaitu berupa serangkaian kode bahasa *assembly*, yang berfungsi sebagai protektor.

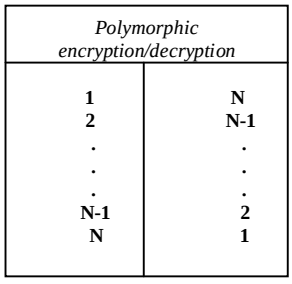
❖ Fungsi Pembangkit dan Pemanggil Kode (Program *Loader*).

- Yaitu berupa prosedur penambah data proteksi ke dalam file PE (target), serta prosedur pemanggil fungsi-fungsi proteksi tersebut.

3.2.1 Fungsi-fungsi Kode Proteksi (Protektor)

Fungsi-fungsi proteksi yang disajikan, dibedakan ke dalam dua bagian utama. Yaitu fungsi-fungsi proteksi yang diberikan secara langsung (tanpa input masukan dari pengguna), serta fungsi-fungsi proteksi yang bersifat opsional (pilihan pengguna).

Fungsi-fungsi proteksi yang diberikan secara langsung (*default*) terdiri atas fungsi enkripsi pada setiap bagian data utama (*Raw Section Data*) serta pada fungsi pembangkitnya (*loader*). Metode enkripsi yang digunakan cukup sederhana, yaitu menggunakan metode ***Polymorphic encryption*** untuk setiap byte kode. Metode ini dibuat oleh seorang teman yang bernama **Danilo Bzdok** serta sudah dalam bentuk file modul (*include file*). Metode enkripsi ini dapat diilustrasikan sebagai berikut :



Gambar 3.2. Ilustrasi Metode Enkripsi Polymorphic

Sedangkan pada fungsi-fungsi proteksi yang bersifat opsional (pilihan pengguna), antara lain adalah sebagai berikut :

❖ **Fungsi Proteksi Anti *Debugger***

Fungsi proteksi ini terutama ditujukan untuk mendeteksi terhadap keberadaan *debugger* SoftIce dan program FrogsICE. FrogsICE tersebut pada umumnya banyak digunakan oleh sejumlah *cracker* yang mengalami kesulitan terhadap beberapa program *shareware* yang memiliki metode pendeteksian SoftIce.

❖ **Fungsi Proteksi Anti *Disassembler***

Tujuan utama dari fungsi proteksi berikut ini adalah untuk mengubah bentuk struktur informasi fungsi impor API yang digunakan saat dilakukan kegiatan *disassembling*. Dengan kata lain fungsi proteksi ini dapat mempersulit seorang *cracker* dalam menganalisa fungsi-fungsi API yang diimpor oleh program tersebut. Dua cara yang dilakukan untuk mengubah struktur informasi fungsi impor API adalah sebagai berikut :

1. Menghapus informasi tabel fungsi impor (*Image Import Descriptors*).
2. Mengalihkan lokasi sesungguhnya fungsi-fungsi API yang diimpor (*Import Address Table*) ke suatu lokasi memori yang lain.

❖ **Fungsi Proteksi Anti Pengubahan Kode (*Code Editing*/*Patching*)**

Tujuan utama dari fungsi proteksi ini adalah agar mencegah terjadinya perubahan data pada keseluruhan badan program. Teknik yang digunakan adalah dengan menggunakan metode ceksum (*checksum*) atau yang lebih dikenal sebagai metode pengecekan **CRC** (***Cyclic Redudancy Code***). Metode pengecekan dengan ceksum pada prinsipnya adalah menghitung total tiap byte data dari suatu file PE.

❖ **Fungsi Proteksi Anti MFD (*Memory Full Dump)***

MFD *(Memory Full Dump*) adalah suatu teknik penyalinan keseluruhan data dari suatu proses atau program yang sedang berjalan di memori. Keseluruhan data tersebut dapat dianggap sebagai badan program *executable* yang telah di-load di memori. Sehingga apabila file PE yang telah terenkripsi (PER) di-load di memori (dieksekusi), maka badan program yang asli (hasil dari dekripsi) dapat diperoleh pada memori. Sehingga dengan teknik MFD dapat diperoleh file PE yang sudah tidak terenkripsi lagi. Contoh program yang dapat melakukan proses MFD adalah **ProcDump32** (G-RoM, Lorian dan Stone) dan **LordPE Deluxe** (Yoda).

Prinsip kerja dari fungsi proteksi ini adalah dengan cara mengelabui program-program seperti ProcDump32 dan LordPE dalam membaca ukuran sebenarnya (*SizeOfImage*) dari proses / program yang ada di memori. Pengubahan tersebut dilakukan dengan cara masuk ke dalam lokasi memori dari variabel internal Windows yang berisi semua ukuran dari proses-proses (*SizeOfImage*) yang sedang berjalan.

❖ **Fungsi Proteksi Anti Pemanggil Proses (*Process Loader*)**

Prinsip kerja dari fungsi proteksi berikut ini adalah mencari informasi proses dari program yang terproteksi pada pusat basis data proses. Dari proses pencarian tersebut akan diperoleh informasi mengenai lingkungan dimana (**ENVIRONMENT_DATABASE**) program tersebut diaktifkan, serta informasi mengenai adanya pemakaian parameter (**CMD_LINE**) saat eksekusi program. Kondisi-kondisi yang mungkin terjadi pada file PE yang diproteksi oleh fungsi

proteksi Anti Pemanggil Proses (*Process Loader*) adalah sebagai berikut :

- Apabila program (file PE) yang diproteksi tersebut dijalankan atau dieksekusi di dalam lingkungan Windows Explorer (file "Explorer.exe"), maka program akan berjalan semestinya.
- Apabila program yang diproteksi tersebut dieksekusi dengan menggunakan suatu fungsi pemanggil seperti fungsi API "**CreateProcess**()", "**WinExec**()", "**ShellExecute**()" maka eksekusi program akan dibatalkan dan berhenti seketika.

❖ **Fungsi Proteksi Penghapus *Header* PE**

Fungsi proteksi ini adalah bertujuan untuk menghapus segala informasi *header* dari struktur file PE pada saat file tersebut selesai di-load di memori. Dengan kata lain informasi-informasi dari struktur ***MS-DOS Header***, ***File Header***, ***Optional Header*** serta ***Section Header*** dihapus saat program telah selesai dimuatkan ke dalam memori. Sedangkan informasi-informasi dari *header* tersebut merupakan komponen sangat penting bagi file PE tersebut.

3.2.2 Fungsi Pembangkit serta Pemanggil Kode (*Loader*)

Fungsi Pembangkit serta Pemanggil Kode, atau yang lebih dikenal sebagai program *loader* merupakan fungsi yang paling utama dari program penginjeksi ini. Hal ini disebabkan oleh karena program *loader* memiliki beberapa tugas penting, yaitu :

1. Berfungsi sebagai komponen yang menggabungkan segala rutin proteksi kedalam satu bagian *Raw Section Data* dari suatu file PE. Dengan kata lain program *loader* berfungsi sebagai penginjeksi atau penambah data pada file PE.
2. Berfungsi sebagai komponen yang menyediakan rutin pemanggil serta pendekripsi dari data-data yang telah dienkripsi dengan metode enkripsi PER (*Polymorphic Encryption / Decryption*).
3. Berfungsi sebagai komponen yang melakukan lompatan ke rutin program sesungguhnya, setelah rutin-rutin proteksi selesai dieksekusi. Maksudnya adalah program *loader* menyediakan ruang pertama kali bagi rutin-rutin proteksi. Baru setelah rutin-rutin proteksi tersebut selesai dieksekusi, program *loader* kembali menuju ke lokasi awal rutin program sesungguhnya (*original entry point* / lokasi awal program sebelum diinjeksi).

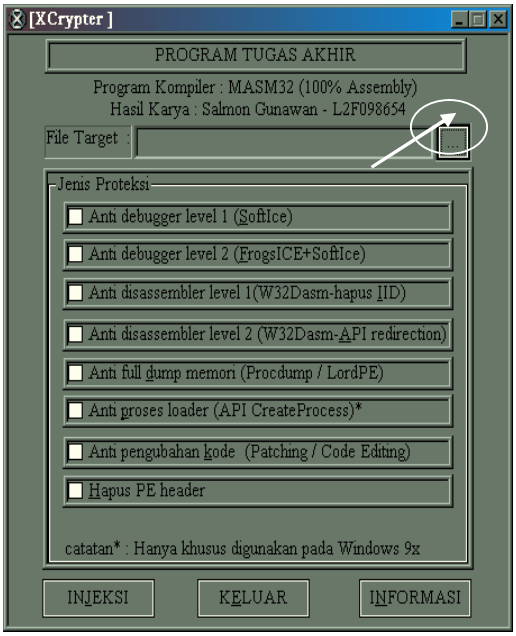
Berikut ini tahap-tahapan algoritma pada saat penginjeksian program *loader* beserta rutin-rutin kode proteksi :

1. Mengakses file target dan mencari informasi yang dibutuhkan untuk identifikasi file PE.
2. Melakukan proses-proses pengecekan terhadap file target.
3. Apabila proses pengecekan berhasil, maka program penginjeksi mengalokasikan suatu ruang di memori yang berfungsi sebagai penampung (*buffer*). *Buffer* tersebut yang nantinya digunakan sebagai ruang untuk mengoperasikan file PE untuk proses-proses selanjutnya. Ukuran dari *buffer* tersebut dicari melalui perhitungan berikut ini :
   BufferOutPutSize = ukuran file PE + ukuran IT (*Import Table*) yang baru + ukuran rutin-rutin kode proteksi (DEPACKER_CODE_SIZE) + ALIGN_CORRECTION (yaitu suatu konstanta bagi kompatibilitas beberapa file PE hasil kompilasi oleh program kompiler WATCOM).
4. Mengambil informasi dari *FileHeader* dan *OptionalHeader* dari file target. Informasi yang diambil dari *FileHeader* adalah jumlah seksi (*section*). Sedangkan informasi yang diambil dari *OptionalHeader* adalah *SizeOfImage* (ukuran file di memori), *AddressOfEntryPoint* (lokasi awal dari eksekusi file target), *ImageBase*. *AddressOfEntryPoint* oleh program loader dinyatakan sebagai OEP (*Original Entry Point*), yaitu lokasi dimulainya eksekusi kode program sesungguhnya. Oleh sebab itu program loader akan mempersiapkan lompatan ke OEP, saat semua rutin proteksi telah dieksekusi.
5. Menyimpan informasi nama-nama file impor *library* beserta fungsi-fungsi API yang diimpor, dari lokasi yang lama menuju ke lokasi IT (*Import Table*) yang baru.
6. Menambah RSD (*Raw Section Data)* yang baru pada direktori seksi (*Section Directory*) file PE. RSD tersebut berfungsi sebagai tempat penyimpanan semua tambahan data, fungsi-fungsi proteksi serta program *loader* dari hasil injeksi. Sedangkan RSD yang baru dibuat tersebut pada *Section Header* diberi nama ".**XCr**".
7. Mengenkripsi semua direktori seksi yang lain.
8. Memperbaharui semua informasi dari *header* file PE.
9. Menghitung ukuran file PE yang baru.
10. Menyalin rutin program *loader* dari memori, untuk persiapan proses penyimpanan.
11. Melakukan proses enkripsi terhadap rutin kode lompatan menuju ke OEP.
12. Mengenkripsi program *loader*.
13. Melakukan perhitungan nilai ceksum dari file PE (produk akhir dari proses injeksi) yang akan disimpan ke dalam *disk*.
14. Menyimpan semua informasi file PE yang di-load di memori ke dalam file sesungguhnya di *disk*.

15. Membersihkan semua alokasi memori dari operasi file PE. Proses penginjeksian selesai dilaksanakan.

IV PENGUJIAN DAN ANALISIS PROGRAM

Tampilan Program XCrpyter



Gambar 4.1. Tampilan Program XCrypter

Dari gambar diatas tampak bahwa program XCrypter memiliki 8 menu pilihan, yaitu antara lain fungsi proteksi anti *debugger* (SoftIce), anti *dissambler* (W32Dasm), anti teknik MFD (*Memory Full Dump*), anti pengubahan kode (*Code Editing*/*Patching*), anti pemanggil proses (*Process Loader*), penghapus *header* file PE.

Keterangan untuk 4 tombol (*button*) yang lain dari program XCrypter adalah sebagai berikut :

- Tombol "**Keluar**" : berfungsi untuk menampilkan pesan konfimasi, yaitu apakah akan keluar dari program XCrypter atau tidak.
- Tombol "**Injeksi**" : yaitu berfungsi untuk melaksanakan proses injeksi pada file PE.
- Tombol "**Informasi**" : yaitu berfungsi untuk menampilkan informasi pembuat dari program XCrypter.
- Tombol yang ditunjukkan oleh anak panah pada gambar 4.1 adalah berfungsi untuk memilih nama file PE yang akan diinjeksikan.

4.2 Pengujian Masing-masing Fungsi Proteksi

Proses pengujian diwakili dengan 5 jenis program executable, dari berbagai hasil kompilasi program kompiler seperti : Borland Delphi, Borland Builder C++, Microsoft Visual C, Microsoft Visual Basic, Assembler 32 (Borland TASM32 atau MASM32). Proses pengujian meliputi dua hal, yaitu :

- **Pengujian pertama** : program hasil injeksi harus dapat berjalan normal (tidak rusak atau cacat).
- **Pengujian kedua** : program hasil injeksi dapat menunjukkan fungsi proteksi yang digunakan dengan cara mengujinya secara langsung dengan program penguji. Program penguji bergantung dengan jenis proteksi yang digunakan. Misalnya pada pengujian fungsi anti *debugger*, maka program pengujinya adalah *debugger* SoftIce.

Proses pengujian yang pertama sangat penting pengaruhnya, oleh karena file PE yang telah diinjeksi akan mengalami proses pengenkripsian data pada bagian pusat data file PE (*Section Directory*). Proses pengenkripsian tersebut bersifat permanen. Sehingga apabila proses pendekripsian tidak berjalan dengan baik, maka dapat dikatakan file PE tersebut cacat/rusak. Apabila hasil eksekusi berjalan dengan normal, maka tahap selanjutnya adalah menganalisa kinerja sistim proteksi yang diberikan, apakah sudah berjalan dengan semestinya. Untuk tahap yang kedua ini, pengujian dilakukan dengan membandingkan antara file PE yang belum terinjeksi dengan yang sudah. Apabila hasil eksekusi menunjukkan ada perbedaan antara keduanya (file yang belum terinjeksi dan yang sudah) , serta telah diuji dengan program penguji seperti **SoftIce**, **URSoft W32Dasm**, **Procdump** serta **LordPE**. Maka dapat dikatakan bahwa proses penginjeksian kode program ke dalam file PE tersebut berjalan dengan semestinya.

Program protektor yang telah dibuat ini bernama **XCrypter**, yaitu kepanjangan dari "***EXE Crypter***". Sedangkan program-program yang akan diujikan tersebut merupakan hasil kompilasi dari berbagai kompiler, yaitu diantaranya :

- Program "**Sysmechanic.exe"** hasil kompilasi program Borland Delphi.
- Program "**mIRC32.exe"** hasil kompilasi program Borland_Builder_C++.
- Program "**IDCrackMe3.exe**" hasil kompilasi program Microsoft Visual Basic.
- Program "**DLLShow.exe**" hasil kompilasi program Microsoft Visual C++.
  Program "**Bin.exe**" hasil kompilasi program MASM32.

V KESIMPULAN DAN SARAN

5.1 Kesimpulan

Dari hasil proses pembuatan, pengujian serta analisa program Tugas Akhir ini, maka dapat ditarik beberapa kesimpulan sebagai berikut :

1. File PE yang telah diinjeksi akan bertambah besar ukurannya sebesar 2 sampai 3 Kbyte, disertai dengan pengenkripsian semua data pada direktori seksi (*Section Directory*).
2. File PE yang telah diinjeksi dapat dilihat hasilnya dengan bertambahnya sebuah direktori seksi (*Section Directory*) bernama "**.XCr**". Direktori seksi tersebut berisi program *loader* beserta rutin-rutin proteksi.
3. File PE yang telah diinjeksi akan memiliki rutin kode eksekusi yang berbeda dibandingkan dengan sebelum diinjeksi. Hal ini disebabkan oleh karena seluruh *Raw Section Data* (hasil injeksi) dienkripsi dengan metode PER (*Polymorphic Encryption*).
4. Hasil pengujian program telah sesuai dengan tujuan yang diharapkan, yaitu dengan terpenuhinya hasil dari masing-masing pengujian (pengujian pertama dan pengujian kedua).
5. Kecepatan proses eksekusi dari file PE hasil injeksi tidak mengalami perubahan yang signifikan oleh karena pertambahan ukuran memori virtual hanya sebesar 2 KB.

5.2 Saran

Program Tugas akhir ini tentunya masih jauh dari sempurna oleh karena keterbatasan penulis. Beberapa masukan dan saran dari penulis yang dapat dikembangkan untuk menuju kesempurnaan dari program **XCrypter** (*Exe Crypter*) adalah sebagai berikut :

1. Fungsi pemroteksi Anti Pemanggil Proses (*Process Loader*) dapat dikembangkan agar dapat berfungsi penuh pada sistim operasi Windows 2000.
2. Fungsi-fungsi proteksi yang dimiliki oleh program XCrypter dapat digabung menjadi satu dengan fungsi kompresi data (*packing*). Sehingga selain mendapatkan fungsi-fungsi proteksi, program-program tersebut juga memiliki ukuran yang lebih kecil. Contoh dari program protektor yang memberikan sistim proteksi dan kompresi data file PE adalah : ASProtect, Petite, PECompact.

DAFTAR PUSTAKA

1. Kath, Randy, ***The Portable Executable File Format from Top to Bottom***, Microsoft Developer Network Technology Group, 1993.
2. Tool Interface Standard (TIS) Formats Specification for Windows Version 1.0, ***Portable Executable (PE) Format***, Microsoft Corporation, 1993.
3. Pietrek, M, ***Peering Inside the PE: A Tour of Win32 Portable Executable File Format***, Microsoft Systems Journal 3, 1994.
4. ……., ***Microsoft Win32 Programmer's Reference***, Microsoft Corporation, 1996.
5. J. O'Leary, Micheal, ***Portable Executable Format***, Microsoft Developer Support, 1997.
6. Philippe Auphelle, ***Win32ASM***, philippe@irci.ihub.com, 1997.
7. Visual C++ Business Unit, ***Microsoft Portable Executable and Common Object File Format Specification***, Microsoft Corporation, Revision 6.0, 1999.
8. Luevelsmeyer, Bernd, ***The PE File Format***, 1999.
9. ……., ***Intel Pentium Instruction Set Reference***, mindweaver@technologist.com, 1999.
10. Brown, Ralf, ***Interrupt List***, Release 61, ralf@pobox.com, 2000.
11. Http://win32asm.cjb.net/index.html.
12. Http://linux20368.dn.net/crackz/Miscpapers.htm.
13. Http://fravia.anticrack.de/

Salmon Gunawan, lahir di Semarang. Telah menyelesaikan studi di SD. Kanisius Hasanudin, SMP Domenico Savio, SMU Kolese Loyola Semarang. Saat ini sedang menyelesaikan Tugas Akhir sebagai syarat meraih gelar Strata-1 (S1) di Teknik Elektro UNDIP.

Mengetahui dan Menyetujui,

Pembimbing II

Sumardi, S.T, M.T
NIP. 132 125 670

Pembimbing I

Ir. Kodrat I.S, M.T
NIP. 132 046 696