

BAB II

LANDASAN TEORI

II.1. Malware

II.1.1. Definisi Malware

Menurut Skoudis dan Zeltser (2003) dalam bukunya *Malware: Fighting Malicious Code*, malware adalah kumpulan instruksi (program) yang berjalan di komputer yang membuat komputer melakukan hal-hal yang dikehendaki penjahat.

II.1.2. Dampak Malware

Dalam definisi di atas, terdapat ungkapan "hal-hal yang dikehendaki penjahat". Dalam bagian yang sama, Skoudis dan Zeltser (2003) menjelaskan bahwa "hal-hal yang dikehendaki penjahat" tersebut praktis adalah semua hal yang dapat dilakukan komputer, termasuk:

- a. menghapus *file-file* penting dari dalam komputer, menyebabkan komputer tidak dapat digunakan,
- b. menulari komputer dan menggunakannya sebagai "batu loncatan" untuk menulari komputer-komputer lainnya yang terhubung ke komputer tersebut,
- c. memonitor ketukan *keyboard*, memungkinkan penjahat untuk mengetahui semua yang diketikkan,
- d. mengumpulkan informasi tentang pengguna komputer, kebiasaan-kebiasaannya, situs-situs web yang dikunjungi, jam *online*-nya, dan lain sebagainya,

- e. mengirimkan tangkapan layar (*screenshot*) komputer ke penjahat, memungkinkan penjahat untuk melihat semua yang ditampilkan di layar,
- f. mengambil gambar dari kamera (atau suara dari mikrofon) yang terpasang di komputer dan mengirimkannya ke penjahat, memungkinkan penjahat untuk mengetahui wajah dan suara pengguna komputer,
- g. menjalankan perintah-perintah penjahat dalam komputer, seperti apabila perintah-perintah tersebut dijalankan langsung oleh pengguna komputer,
- h. mencuri *file-file* dari dalam komputer, khususnya *file-file* penting yang berisi informasi pribadi, keuangan, dan sebagainya,
- i. mengunggah (*upload*) *file-file* ke dalam komputer, seperti *malware-malware* lainnya, data curian, perangkat lunak bajakan, pornografi, menjadikan komputer sebagai tempat penyediaan (*hosting*) barang-barang ilegal,
- j. menggunakan komputer sebagai "batu loncatan" untuk menyerang komputer lainnya, memalsukan sumber serangan dan membingungkan pihak berwajib,
- k. menuduh pengguna komputer atas kejahatan yang tidak pernah dilakukannya, dengan menanam bukti-bukti palsu dalam komputer, dan
- l. menyembunyikan aktivitas-aktivitas jahat dalam komputer dengan menyembunyikan *file-file*, proses-proses, dan aktivitas jaringan.

II.1.3. Alasan "Mewabahnya" *Malware*

Menurut Skoudis dan Zeltser (2003), ada beberapa hal yang menyebabkan "mewabahnya" *malware* hingga menjadi "wabah besar" seperti sekarang, yaitu:

- a. **Aplikasi-aplikasi mencampur data dan instruksi dalam dokumen:** Dengan mencampur data dan instruksi dalam dokumen, aplikasi-aplikasi menambah jalan masuk *malware* ke dalam komputer. Hal ini diperparah dengan rata-rata pengguna komputer yang tidak mengerti instruksi-instruksi semacam itu dan anggapan rata-rata pengguna komputer bahwa membuka dokumen "pasti aman" karena "hanya dokumen". Sayangnya, aplikasi-aplikasi semacam itu malah semakin digemari, karena interaktivitas, fleksibilitas, dan efisiensi yang ditawarkannya.
- b. **Aplikasi-aplikasi dirancang tanpa memperhatikan faktor keamanan:** Pada zaman dahulu, aplikasi-aplikasi dirancang tanpa memperhatikan faktor keamanan: aplikasi-aplikasi dirancang dengan asumsi hanya akan digunakan dalam lingkungan yang "bersahabat". Selama beberapa waktu, asumsi tersebut memang benar. Sayangnya, pada zaman sekarang, dengan masalah *malware* yang sedemikian parahnya, asumsi tersebut hancur: faktor keamanan tidak bisa lagi diabaikan dalam perancangan aplikasi. Sayangnya, pemeriksaan milyaran baris program dalam ribuan aplikasi yang ada tidak dapat dilakukan dalam waktu singkat. Akibatnya, muncullah masalah *malware* dalam aplikasi-aplikasi zaman sekarang.
- c. **Lingkungan komputer semakin homogen:** Pada zaman dahulu, komputer-komputer berjalan pada *platform* perangkat keras, perangkat lunak, dan jaringan yang berbeda-beda. Akibatnya, penyebaran *malware* terbatas ke beberapa komputer yang berjalan pada *platform* yang sama saja. Sayangnya, pada zaman sekarang, sebagian besar komputer berjalan pada beberapa *platform* perangkat keras (x86 dan x64), perangkat lunak

(Windows, Linux, dan Macintosh), dan jaringan (Ethernet dan TCP/IP) saja. Akibatnya, sebuah *malware* dapat menyebar ke sejumlah besar komputer.

- d. **Konektivitas semakin berkembang:** Pada zaman dahulu, hanya sedikit komputer yang terhubung ke jaringan, dan skala jaringannya pun masih sangat terbatas. Akibatnya, penyebaran *malware* terbatas ke beberapa komputer yang terhubung saja. Sayangnya, pada zaman sekarang, hampir setiap alat menjadi berbasis komputer, terhubung ke Internet, dan menggunakan Ethernet dan protokol TCP/IP. Akibatnya, sebuah *malware* dapat menyebar ke sejumlah besar komputer dalam waktu singkat.
- e. **Pengguna pemula jumlahnya semakin banyak:** Pada zaman dahulu, komputer-komputer hanya digunakan oleh para "pakar komputer" yang benar-benar mengerti komputer. Akibatnya, aplikasi-aplikasi pada zaman itu dirancang dengan asumsi penggunaanya benar-benar mengerti apa yang akan dilakukannya (termasuk tindakan-tindakan beresiko). Meskipun demikian, hal tersebut lambat laun berubah, sampai pada zaman sekarang, dimana sebagian besar komputer digunakan oleh orang yang tidak mengerti komputer. Sayangnya, masih banyak aplikasi yang belum beradaptasi dengan perubahan tersebut, memunculkan aplikasi-aplikasi yang memungkinkan tindakan beresiko tanpa konfirmasi, memungkinkan para penggunaanya untuk memasukkan *malware* ke dalam komputer mereka.
- f. **Dunia menjadi tempat yang tidak aman:** Dalam beberapa tahun terakhir, berita-berita internasional menunjukkan bahwa dunia, yang dulunya dianggap sebagai tempat yang aman, menjadi tempat yang tidak aman lagi. Berita-berita tentang peperangan, pemberontakan,

perang saudara, terorisme, sengketa, dan kejahatan mengisi berita-berita internasional setiap harinya. Lambat laun, hal ini tentu akan menyebar ke ranah teknologi informasi, dalam bentuk perang Internet (*cyberwar*), terorisme Internet (*cyberterrorism*), dan kejahatan Internet (*cybercrime*).

II.1.4. Jenis *Malware*

Menurut Skoudis dan Zeltser (2003), ada tujuh jenis *malware* yang dikenal, yaitu:

- a. **Virus**, merupakan program komputer yang menulari *file-file* dengan menyisipkan salinan dirinya dalam *file-file* tersebut. Salinan-salinan tersebut biasanya berjalan ("aktif") saat *file-file* yang berisi salinan dimuat ke memori (biasanya saat dibuka atau dijalankan), memungkinkan virus untuk menulari *file-file* lainnya.
- b. **Worm**, merupakan program komputer yang menyebarkan dirinya ke komputer-komputer, biasanya dengan membuat salinan dirinya dalam memori komputer-komputer tersebut. Sebuah *worm* dapat terlalu sering menggandakan dirinya dalam sebuah komputer sehingga menyebabkan komputer tersebut menjadi *crash*.
- c. **Malicious mobile code**, merupakan virus atau *malware* lain yang memanfaatkan celah keamanan dalam sistem transmisi nirkabel. *Malicious mobile code* dapat mempengaruhi komputer, PDA, telepon seluler, dan perangkat komunikasi nirkabel lainnya.
- d. **Backdoor**, merupakan program komputer yang mengizinkan akses ke dalam sebuah sistem komputer tanpa melalui sistem keamanannya.
- e. **Trojan horse**, merupakan program berbahaya yang disamarkan sebagai program permainan, utilitas, atau

aplikasi. Saat dijalankan, sebuah *trojan horse* tampak melakukan hal bermanfaat, padahal sebenarnya sedang melakukan hal berbahaya.

- f. **Rootkit**, merupakan *trojan horse* yang memodifikasi sistem operasi tempat dirinya terinstal untuk menyembunyikan dirinya, misalnya mencegah tampilnya dirinya dalam daftar proses atau *file* sistem operasi.
- g. **Malware kombinasi**, merupakan *malware* yang memiliki teknik (dapat dikelompokkan dalam) lebih dari satu jenis *malware* di atas.

II.1.5. Sejarah Malware

Menurut Skoudis dan Zeltser (2003), meskipun *malware* baru menjadi "wabah besar" belakangan ini, *malware* memiliki sejarah panjang, hampir sepanjang sejarah komputer pribadi, sebagai berikut:

- a. **1981 - 1982 - Virus komputer pertama**: Paling tidak tiga virus, termasuk Elk Cloner, ditemukan dalam program-program permainan untuk komputer Apple II, meskipun kata *virus* saat itu belum dikenal.
- b. **1983 - Definisi formal virus komputer**: Fred Cohen mendefinisikan virus komputer sebagai "program komputer yang dapat menulari program-program komputer lainnya dengan memodifikasi mereka agar mencantumkan salinan, yang mungkin telah 'berevolusi', dari dirinya."
- c. **1986 - Virus PC pertama**: Virus Brain menulari sistem Microsoft DOS, sebuah pertanda serangan-serangan *malware* di masa yang akan datang, dimana sistem operasi DOS (kemudian Windows) menjadi sasaran utama virus-virus dan *worm-worm*.
- d. **1988 - Worm Morris**: Ditulis oleh Robert Tappan Morris, Jr. dan diluncurkan pada bulan November, *worm* awal ini

mematikan sebagian besar Internet pada masa itu dan menjadi berita utama di seluruh dunia.

- e. **1990 – Virus *polymorphic* pertama:** Untuk menghindari antivirus, virus-virus *polymorphic* mengubah penampilannya setiap kali dijalankan, membuka bidang penelitian kode *polymorphic* yang masih terus diteliti hingga sekarang.
- f. **1991 – Virus Construction Set (VCS) diluncurkan:** Pada bulan Maret, peralatan ini diluncurkan ke komunitas *bulletin board system*, memungkinkan para penulis virus pemula untuk menciptakan virusnya sendiri.
- g. **1994 – Hoax virus Good Times:** "Virus" ini sebenarnya tidak ada dan hanya merupakan berita bohong (*hoax*) belaka. Meskipun demikian, kabar akan "virus" ini menyebar dari orang ke orang, menyebabkan kekhawatiran.
- h. **1995 – Virus *macro* pertama:** Jenis virus ini diimplementasikan dalam bahasa *macro* Microsoft Word, menulari *file-file* dokumennya. Teknik ini kemudian juga digunakan dalam bahasa *macro* dalam aplikasi-aplikasi lainnya.
- i. **1996 – Netcat diluncurkan untuk UNIX:** Peralatan yang ditulis oleh Hobbit ini tetap menjadi *backdoor* paling populer untuk sistem-sistem UNIX hingga saat ini. Meskipun Netcat memiliki banyak kegunaan, baik yang legal maupun ilegal, Netcat paling sering di(salah)gunakan sebagai *backdoor*.
- j. **1998 – Virus Java pertama:** Virus StrangeBrew menulari *applet-applet* Java lainnya, membawa kekhawatiran akan virus ke ranah aplikasi web.
- k. **1998 – Netcat diluncurkan untuk Windows:** Windows rupanya juga tidak terlindung terhadap Netcat. Ditulis

oleh Weld Pond, Netcat juga digunakan sebagai *backdoor* populer dalam sistem-sistem Windows.

- l. **1998 – Back Orifice:** Peralatan yang diluncurkan pada bulan Juli oleh Cult of the Dead Cow (cDc), sebuah kelompok *hacking*, ini memungkinkan pengendalian jarak jauh (*remote control*) sistem-sistem Windows melalui jaringan.
- m. **1999 – Virus/worm Melissa:** Diluncurkan pada bulan Maret, virus *macro* Microsoft Word ini menulari ribuan komputer di seluruh dunia melalui *e-mail*. Melissa adalah virus sekaligus *worm*, karena selain menulari *file-file* dokumen, juga menyebar lewat jaringan.
- n. **1999 – Back Orifice 2000 (BO2K):** Pada bulan Juli, cDc meluncurkan versi Back Orifice yang telah ditulis dari awal, yang berfungsi mengendalikan sistem Windows dari jarak jauh (*remote control*). Versi baru ini memiliki antarmuka *point-and-click*, *Application Programming Interface* (API), dan pengendalian *mouse*, *keyboard*, serta *layar*.
- o. **1999 – Agen distributed denial of service:** Pada akhir musim panas, agen *denial of service* Tribe Flood Network (TFN) dan Trin00 diluncurkan. Peralatan-peralatan ini memungkinkan seorang penjahat untuk mengendalikan ribuan komputer yang memiliki agen dari sebuah komputer. Dengan pengendalian terpusat, agen-agen tersebut dapat menjalankan serangan *flood* yang mematikan.
- p. **1999 – Knark Kernel-Level RootKit:** Pada bulan November, seseorang bernama Creed meluncurkan peralatan manipulasi *kernel* Linux ini. Knark memiliki peralatan lengkap untuk memanipulasi *kernel* Linux sehingga seorang penjahat dapat menyembunyikan *file-file*, proses-proses, dan aktivitas jaringan.

- q. **2000 – Love Bug:** Pada bulan Mei, worm VBScript ini mematikan puluhan ribu sistem di seluruh dunia dengan menyebar lewat beberapa celah keamanan Microsoft Outlook.
- r. **2001 – Worm Code Red:** Pada bulan Juli, worm ini menyebar lewat celah keamanan *buffer overflow* dalam *web server* Microsoft IIS. Lebih dari 250.000 komputer menjadi korban dalam waktu kurang dari delapan jam.
- s. **2001 – Kernel Intrusion System:** Juga pada bulan Juli, peralatan yang dikembangkan oleh Optyx ini memudahkan manipulasi *kernel* Linux dengan menyediakan antarmuka grafis (GUI) yang mudah digunakan dan kemampuan persembunyian yang sangat efektif.
- t. **2001 – Worm Nimda:** Hanya seminggu setelah serangan teroris 11 September, worm yang sangat menular ini menggunakan banyak metode untuk menulari sistem-sistem Windows, termasuk *exploit buffer overflow web server*, *exploit web browser*, serangan melalui *e-mail* Outlook, dan *file sharing*.
- u. **2002 – Backdoor Setiri:** Meskipun tidak pernah diluncurkan secara resmi, *trojan horse* ini dapat melewati *firewall personal*, *firewall* jaringan, dan perangkat-perangkat *Network Address Translation* dengan menyalahgunakan (jendela) *web browser* yang tidak terlihat.
- v. **2003 – Worm SQL Slammer:** Pada bulan Januari, worm ini menyebar dengan cepat, mematikan beberapa penyedia jasa Internet di Korea Selatan dan, selama beberapa saat, menyebabkan masalah-masalah di seluruh dunia.
- w. **2003 – Hydan Executable Steganography Tool:** Diluncurkan pada bulan Februari, peralatan ini memungkinkan penyembunyian data dengan teknik kode *polymorphic* dalam *file-file executable* Linux, BSD, dan

Windows. Konsep ini juga dapat dikembangkan untuk menghindari antivirus dan *intrusion detection system*.

II.1.6. Perkembangan Malware

Berdasarkan sejarah *malware* di atas, Skoudis dan Zeltser (2003) menyimpulkan beberapa "tren" perkembangan *malware*, sebagai berikut:

- a. **Meningkatnya kompleksitas malware:** Dari virus-virus Apple II yang cukup sederhana hingga peralatan-peralatan manipulasi *kernel* yang kompleks dan *worm-worm* yang ampuh pada zaman sekarang, *malware-malware* semakin pintar dalam menularkan dan menyembunyikan dirinya.
- b. **Makin cepatnya pertumbuhan peralatan-peralatan dan teknik-teknik malware:** Konsep-konsep baru *malware* biasanya tumbuh dengan lambat, tetapi semakin cepat seiring waktu. Dalam lima tahun terakhir, peralatan-peralatan dan teknik-teknik *malware* baru semakin canggih dan semakin cepat pertumbuhannya, dan tren ini hanya akan terus meningkat.
- c. **Pergerakan dari virus ke worm ke eksploitasi kernel:** Pada zaman dahulu, kebanyakan aksi *malware* berputar sekitar virus-virus yang menulari program-program komputer. Dalam lima tahun terakhir, fokus *malware* bergerak ke *worm-worm* dan eksploitasi *kernel-kernel* sistem operasi.

II.2. DLL Injection dan API Hooking

Dalam Windows, setiap proses memiliki ruang alamatnya (*address space*) masing-masing. Ketika penulis menggunakan *pointer* untuk merujuk memori, *pointer* tersebut merujuk alamat memori dalam ruang alamat proses penulis (penulis tidak dapat menciptakan *pointer* yang

merujuk memori proses lainnya). Jadi, jika proses penulis mengandung *bug* yang menimpa memori secara acak, *bug* tersebut tidak akan mempengaruhi memori proses lainnya.

Pemisahan ruang alamat (*address space separation*) memberikan manfaat besar bagi para pengembang dan pengguna. Bagi para pengembang, pemisahan ruang alamat membantu mencegah *bug* yang mengakses memori secara acak. Sementara itu, bagi para pengguna, pemisahan ruang alamat membuat sistem operasi menjadi lebih handal karena sebuah aplikasi tidak dapat mematikan proses-proses lainnya (atau sistem operasi sendiri). Sayangnya, pemisahan ruang alamat juga memiliki sisi negatifnya, yaitu akan lebih sulit menulis aplikasi yang dapat memanipulasi atau berkomunikasi dengan proses lainnya.

Keadaan-keadaan yang memerlukan pengaksesan memori proses lain antara lain adalah:

- a. *subclassing* jendela milik proses lain,
- b. bantuan *debugging* (misalnya untuk mengetahui DLL mana saja yang digunakan proses lain), dan
- c. *hooking* proses lain.

Dalam bagian ini, penulis akan menunjukkan berbagai cara menginjeksikan DLL ke ruang alamat proses lain. Saat DLL penulis telah berada dalam ruang alamat proses lain, penulis dapat menjalankan kode dalam ruang alamat proses lain tersebut sesuai kehendak.

II.2.1. Contoh: *Subclassing* Jendela Proses Lain

Sebagai contoh, andaikan penulis ingin men-*subclass* sebuah jendela milik proses lain. Untuk melakukan hal itu, penulis memanggil fungsi `SetWindowLongPtr` untuk mengubah alamat prosedur jendela (*window procedure*) dalam struktur memori jendela tersebut agar mengarah ke prosedur jendela penulis. Dokumentasi Platform SDK

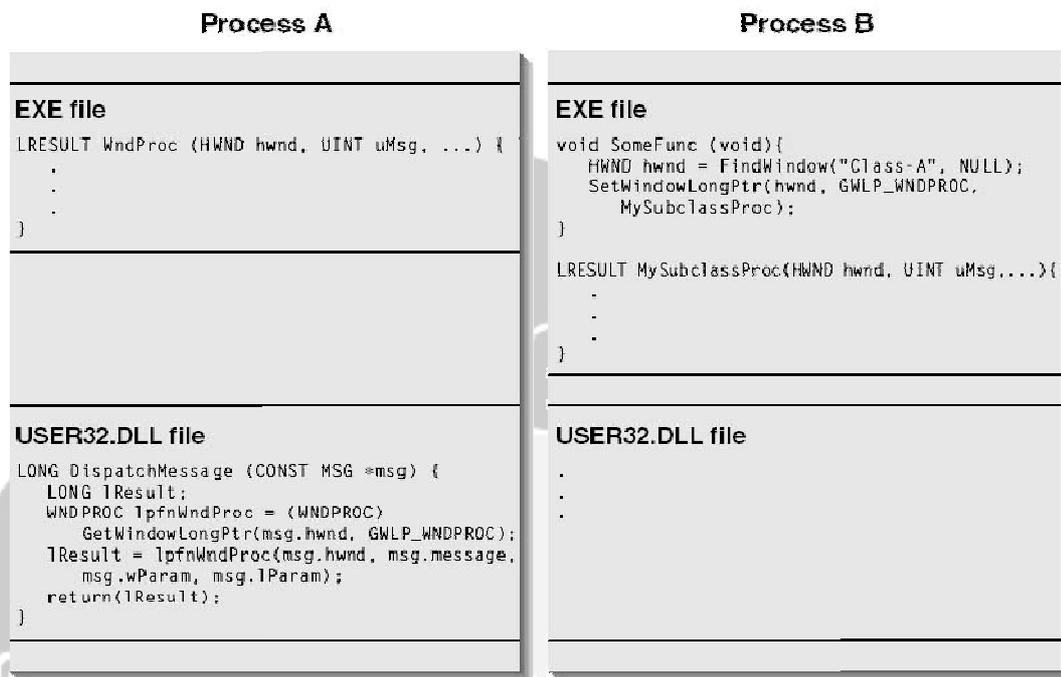
menyatakan bahwa sebuah aplikasi tidak dapat men-*subclass* jendela milik proses lainnya. Hal tersebut tidak dapat dilakukan bukan karena sekedar dilarang, melainkan karena berkaitan dengan pemisahan ruang alamat proses.

Ketika penulis memanggil `SetWindowLongPtr` untuk men-*subclass* sebuah jendela, seperti berikut, penulis memberitahu sistem bahwa semua pesan (*message*) yang dikirimkan (*sent*) atau dipasangkan (*posted*) ke jendela yang dirujuk `hwnd` harus ditujukan ke `MySubclassProc`, bukan prosedur asli jendela tersebut.

```
SetWindowLongPtr( hwnd, GWLP_WNDPROC, MySubclassProc );
```

Dengan kata lain, ketika sistem akan mengantarkan (*dispatch*) sebuah pesan ke prosedur jendela tersebut, sistem akan mencari alamat prosedur jendela tersebut lalu memanggilnya. Dalam contoh di atas, sistem melihat bahwa alamat fungsi `MySubclassProc` berkaitan dengan jendela tersebut, sehingga memanggil `MySubclassProc`.

Permasalahan *subclassing* jendela milik proses lain adalah bahwa prosedur *subclass* ada dalam ruang alamat proses lainnya. Gambar 2.1. menunjukkan bagaimana sebuah prosedur jendela menerima pesan. Proses A sedang berjalan dan memiliki sebuah jendela. *File* `User32.dll` dipetakan (*mapped*) dalam ruang alamat proses A. Pemetaan `User32.dll` ini bertanggungjawab menerima dan mengantarkan pesan-pesan untuk jendela-jendela milik proses A. Ketika pemetaan `User32.dll` ini mendeteksi sebuah pesan, pertama-tama pemetaan tersebut akan menentukan alamat prosedur jendela tujuan pesan lalu memanggilnya, dengan melewati (*pass*) *handle* jendela, pesan, dan nilai `wParam` serta `lParam`. Setelah prosedur jendela memproses pesan tersebut, `User32.dll` akan kembali dan menunggu pesan berikutnya untuk diproses.



Gambar 2.1. Proses B mencoba men-*subclass* jendela milik proses A

Sekarang andaikan proses penulis adalah proses B dan penulis ingin men-*subclass* jendela milik proses A. Kode penulis dalam proses B pertama-tama harus menentukan *handle* jendela yang ingin di-*subclass*. Ini dapat dilakukan dengan berbagai cara. Contoh yang ditampilkan dalam gambar 2.1. memanggil fungsi `FindWindow` untuk memperoleh *handle* jendela yang diinginkan. Selanjutnya, proses B memanggil `SetWindowLongPtr` untuk mengubah alamat prosedur jendela tersebut. Panggilan ini gagal (menghasilkan `NULL`), karena kode dalam `SetWindowLongPtr` mencegah sebuah proses mengubah alamat prosedur jendela milik proses lainnya dan menolak permintaan tersebut.

Bagaimana jika fungsi `SetWindowLongPtr` dapat mengubah alamat prosedur jendela tersebut? Dalam hal itu, sistem akan mengaitkan alamat `MySubclassProc` dengan jendela tersebut. Selanjutnya, ketika jendela tersebut

dikirim pesan, kode USER32 dalam proses A akan menerima pesan, memperoleh alamat MySubclassProc, lalu memanggilnya. Akan tetapi, MySubclassProc ada dalam ruang alamat proses B, padahal pemanggilan dilakukan oleh proses A. Seandainya USER32 memanggil alamat tersebut, pemanggilan akan dilakukan dalam ruang alamat proses A, dan ini akan menghasilkan pelanggaran akses memori (*access violation*).

Untuk menghindari masalah ini, misalkan penulis ingin agar sistem mengetahui bahwa MySubclassProc ada dalam ruang alamat proses B dan agar sistem mengubah proses aktif (*context switching*) sebelum memanggil prosedur *subclass*. Microsoft tidak mengimplementasikan kemampuan ini karena beberapa alasan:

- a. Aplikasi-aplikasi jarang men-*subclass* jendela milik proses lain. Kebanyakan aplikasi men-*subclass* jendela miliknya sendiri, dan arsitektur memori Windows tidak melarang hal ini.
- b. Perubahan proses aktif memerlukan waktu yang lama.
- c. Sebuah *thread* dalam proses B harus menjalankan kode MySubclassProc. *Thread* manakah yang harus digunakan? *Thread* yang ada atau *thread* baru?
- d. Bagaimana caranya User32.dll mengetahui apakah alamat yang berkaitan dengan sebuah jendela merupakan alamat prosedur dalam proses lain atau proses yang sama?

Karena tidak ada solusi yang memuaskan atas masalah-masalah di atas, Microsoft memutuskan untuk tidak mengizinkan SetWindowLongPtr untuk mengubah prosedur jendela milik proses lainnya.

Akan tetapi, ini bukan berarti *subclassing* jendela milik proses lain tidak dapat dilakukan. Hal tersebut

masih dapat dilakukan, meskipun dengan cara yang berbeda. Pertanyaannya di sini sebenarnya bukanlah tentang *subclassing*, melainkan tentang pemisahan ruang alamat. Jika penulis dapat membawa kode prosedur *subclass* penulis ke ruang alamat proses A, penulis dapat memanggil *SetWindowLongPtr* dan melewati alamat *MySubclassProc* dalam proses A. Teknik ini disebut *DLL injection* ke ruang alamat proses, dan dapat dilakukan dalam beberapa cara.

II.2.2. *DLL Injection* Melalui *Registry*

DLL injection dapat dilakukan melalui *registry* dengan memodifikasi entri *registry* berikut:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\  
Windows NT\CurrentVersion\Windows\AppInit_DLLs
```

Entri tersebut dapat berisi sebuah nama *file* DLL atau sejumlah nama *file* DLL yang dipisahkan spasi atau koma. Karena nama-nama *file* dipisahkan spasi, nama-nama *file* yang mengandung spasi harus dihindari. Nama *file* DLL pertama dapat mengandung *path*, sementara nama-nama berikutnya yang mengandung *path* akan diabaikan. Oleh karena itu, sebaiknya *file* DLL ditempatkan dalam direktori sistem Windows agar tidak memerlukan *path*.

Ketika komputer di-*restart* dan Windows mulai berjalan, sistem akan menyimpan nilai entri tersebut. Setelah itu, ketika pustaka *User32.dll* dipetakan dalam sebuah proses, pustaka *User32.dll* akan menerima pemberitahuan (*notification*) *DLL_PROCESS_ATTACH*. Saat pemberitahuan tersebut diproses, *User32.dll* akan membaca nilai entri yang telah disimpan dan memanggil *LoadLibrary* untuk setiap DLL yang disebutkan dalam entri. Sebagaimana setiap pustaka dimuat, *DllMain* setiap pustaka akan dipanggil dengan *fdwReason* bernilai *DLL_PROCESS_ATTACH*, sehingga setiap pustaka dapat melakukan inisialisasi.

Karena DLL yang diinjeksikan dimuat pada awal masa hidup proses, fungsi-fungsi yang dipanggil DLL tersebut harus diperhatikan: fungsi-fungsi `Kernel32.dll` dapat dipanggil dengan aman, sementara pemanggilan fungsi-fungsi DLL lainnya dapat menimbulkan masalah. Selain itu, `User32.dll` juga tidak akan memeriksa apakah setiap pustaka telah dimuat dan diinisialisasi dengan sukses atau tidak.

Dari semua metode *DLL injection*, metode ini merupakan yang termudah. Satu-satunya yang perlu dilakukan adalah menambahkan nilai ke sebuah entri *registry*. Sayangnya, teknik ini juga memiliki beberapa kelemahan:

- a. Karena sistem hanya membaca entri *registry* tersebut pada saat mulai (*startup*), komputer harus di-*restart* setelah mengubah entri *registry* tersebut. Demikian pula, jika sebuah DLL dihapus dari entri *registry* tersebut, komputer tidak akan berhenti memetakan pustaka tersebut sampai komputer di-*restart*.
- b. DLL yang diinjeksikan hanya akan dipetakan dalam proses yang menggunakan `User32.dll`. Meskipun semua aplikasi GUI menggunakan `User32.dll`, kebanyakan aplikasi *command line* tidak. Jadi, metode ini tidak cocok digunakan dalam aplikasi *command line*.
- c. DLL yang diinjeksikan akan dipetakan dalam setiap aplikasi GUI, meski hanya beberapa proses saja yang perlu diinjeksi. Sementara itu, semakin banyak proses yang memetakan DLL tersebut, semakin besar pula kemungkinan proses-proses tersebut untuk *crash* (karena setiap *thread* proses-proses tersebut akan menjalankan DLL tersebut). Jika DLL yang diinjeksikan *hang* atau

mengakses memori secara keliru, proses-proses yang memetakan DLL tersebut juga akan terpengaruh. Oleh karena itu, sebaiknya DLL diinjeksikan ke sesedikit mungkin proses.

- d. DLL yang diinjeksikan akan dipetakan dalam setiap aplikasi GUI sepanjang hidupnya. Masalah ini hampir sama dengan masalah sebelumnya. Idealnya, DLL yang diinjeksikan seharusnya dipetakan dalam proses-proses yang perlu diinjeksi saja, dalam waktu sesingkat mungkin. Misalkan seorang pengguna menjalankan aplikasi yang *men-subclass* jendela utama WordPad. DLL aplikasi tersebut tidak perlu dipetakan dalam ruang alamat WordPad sampai pengguna menjalankan aplikasi. Jika kemudian pengguna keluar dari aplikasi, jendela utama WordPad seharusnya tidak lagi *di-subclass* (*unsubclassed*). Dalam keadaan ini, DLL aplikasi tidak perlu lagi diinjeksikan ke ruang alamat WordPad.

II.2.3. DLL Injection Melalui Windows Hook

DLL injection juga dapat dilakukan melalui *hook*. Agar *hook* dapat bekerja seperti dalam Windows 16-bit, Microsoft mengembangkan mekanisme yang memungkinkan sebuah DLL untuk diinjeksikan ke ruang alamat proses lain.

Sebagai contoh, andaikan proses A (sebuah utilitas seperti Microsoft Spy++) memasang *hook* `WH_GETMESSAGE` untuk melihat pesan-pesan yang diproses jendela-jendela dalam sistem. *Hook* tersebut dipasang dengan memanggil `SetWindowsHookEx`, sebagai berikut:

```
HHOOK hHook = SetWindowsHookEx( WH_GETMESSAGE, GetMsgProc,
                                hinstDll, 0);
```

Parameter pertama, `WH_GETMESSAGE`, menentukan jenis *hook* yang ingin dipasang. Parameter kedua, `GetMsgProc`, mengenali alamat fungsi (dalam ruang alamat proses pemanggil) yang harus dipanggil sistem ketika sebuah jendela akan memproses sebuah pesan. Parameter ketiga, `hinstDll`, mengenali DLL yang mengandung fungsi `GetMsgProc`. Dalam Windows, nilai `hinstDll` sebuah DLL mengenali alamat memori *virtual* (*virtual memory address*) tempat DLL dipetakan dalam ruang alamat proses. Parameter terakhir, `0`, menentukan *thread* yang ingin di-*hook*. Sebuah *thread* bisa saja memanggil `SetWindowsHookEx` dan melewati ID *thread* lainnya dalam sistem. Dengan melewati `0` untuk parameter tersebut, sistem diberitahu untuk meng-*hook* semua *thread* GUI dalam sistem.

Setelah fungsi `SetWindowsHookEx` dipanggil, hal-hal yang akan terjadi adalah sebagai berikut:

1. Sebuah *thread* dalam proses B akan mengantarkan sebuah pesan ke sebuah jendela.
2. Sistem memeriksa apakah ada *hook* `WH_GETMESSAGE` yang terpasang pada *thread* tersebut.
3. Sistem memeriksa apakah DLL yang mengandung fungsi `GetMsgProc` telah dipetakan dalam ruang alamat proses B.
4. Jika DLL tersebut belum dipetakan, sistem akan memetakan DLL tersebut dalam ruang alamat proses B dan menaikkan (*increment*) *lock count* DLL tersebut.
5. Sistem memeriksa apakah `hinstDll` DLL tersebut dalam proses B sama dengan `hinstDll` DLL tersebut dalam proses A.
 - a. Jika kedua `hinstDll` sama, berarti alamat memori fungsi `GetMsgProc` sama antar kedua proses. Dalam keadaan ini, sistem dapat

langsung memanggil fungsi `GetMsgProc` dalam ruang alamat proses A.

- b. Jika kedua `hinstDll` berbeda, sistem harus menghitung alamat memori *virtual* fungsi `GetMsgProc` dalam ruang alamat proses B. Alamat ini dihitung dengan rumus berikut:

$$\text{GetMsgProc B} = \text{hinstDll B} + (\text{GetMsgProc A} - \text{hinstDll A})$$

Pengurangan `hinstDll A` dari `GetMsgProc A` akan menghasilkan offset fungsi `GetMsgProc`. Penambahan offset ini ke `hinstDll B` akan menghasilkan alamat fungsi `GetMsgProc` dalam ruang alamat proses B.

6. Sistem menaikkan *lock count* DLL tersebut.
7. Sistem memanggil fungsi `GetMsgProc` dalam ruang alamat proses B.
8. Saat `GetMsgProc` kembali, sistem menurunkan (*decrement*) *lock count* DLL tersebut.

Perhatikan bahwa saat sistem menginjeksikan atau memetakan DLL yang mengandung fungsi *hook*, keseluruhan DLL akan dipetakan, bukan hanya fungsi *hook*-nya saja. Ini berarti semua fungsi DLL tersebut dapat dipanggil dari proses B.

Jadi, untuk men-*subclass* jendela milik proses lain, pertama-tama penulis memasang *hook* `WH_GETMESSAGE` pada *thread* yang menciptakan jendela tersebut. Setelah itu, saat fungsi `GetMsgProc` dipanggil, penulis memanggil `SetWindowLongPtr` untuk men-*subclass* jendela tersebut. Tentu saja, prosedur *subclass* harus berada dalam DLL yang sama dengan fungsi `GetMsgProc`.

Tidak seperti metode *DLL injection* melalui *registry*, metode ini memungkinkan DLL untuk tidak lagi dipetakan

(*unmapped*) saat DLL tersebut tidak lagi diperlukan dalam ruang alamat proses lain, dengan memanggil:

```
BOOL UnhookWindowsHookEx( HHOOK hhook );
```

Ketika sebuah *thread* memanggil fungsi `UnhookWindowsHookEx`, sistem akan menurunkan *lock count* DLL tempat fungsi *hook* berada. Saat *lock count* mencapai 0, DLL tersebut secara otomatis akan tidak dipetakan dalam ruang alamat proses. Langkah ini dilakukan karena sebelum sistem memanggil fungsi `GetMsgProc`, sistem telah menaikkan *lock count* DLL tersebut (langkah 6). Langkah ini diperlukan untuk menghindari pelanggaran akses memori: jika *lock count* tidak dinaikkan, *thread* lain dalam sistem bisa saja memanggil `UnhookWindowsHookEx` saat *thread* proses B sedang menjalankan fungsi `GetMsgProc`.

Secara keseluruhan, ini berarti sebuah *hook* tidak bisa dilepas segera setelah jendela selesai di-*subclass*. *Hook* tersebut harus terpasang sepanjang hidup *subclass*.

II.2.4. *DLL Injection Melalui Remote Thread*

Metode ketiga *DLL injection*, melalui *remote thread*, merupakan yang paling fleksibel. Kebanyakan fungsi Windows hanya memungkinkan sebuah proses untuk memanipulasi dirinya sendiri. Akan tetapi, ada beberapa fungsi yang memungkinkan sebuah proses untuk memanipulasi proses lainnya. Kebanyakan fungsi tersebut pada mulanya dirancang untuk *debugger* dan alat-alat lainnya. Meskipun demikian, semua aplikasi dapat memanggil fungsi-fungsi tersebut.

Pada dasarnya, teknik *DLL injection* ini memaksa sebuah *thread* dalam proses tujuan untuk memanggil `LoadLibrary` untuk memuat DLL yang diinginkan. Karena penulis tidak dapat mengendalikan *thread-thread* dalam

proses lain dengan mudah, teknik ini memerlukan penciptaan *thread* baru dalam proses tujuan. Karena *thread* ini diciptakan penulis sendiri, penulis dapat mengendalikan kode-kode yang dijalankannya. Untungnya, Windows memiliki fungsi `CreateRemoteThread` yang memungkinkan penciptaan *thread* dalam proses lain:

```
HANDLE CreateRemoteThread(
    HANDLE hProcess,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD fdwCreate,
    PDWORD pdwThreadId);
```

`CreateRemoteThread` sama dengan `CreateThread`, kecuali bahwa `CreateRemoteThread` memiliki sebuah parameter tambahan, yaitu `hProcess`. Parameter ini mengenali proses yang akan memiliki *thread* baru. Parameter `pfnStartAddr` mengenali alamat memori fungsi *thread*. Tentu saja, alamat memori ini relatif terhadap ruang alamat proses yang ingin diinjeksi – kode fungsi *thread* tidak boleh berada dalam ruang alamat proses penulis.

Setelah menciptakan *thread* dalam proses lain, penulis kemudian harus memaksa *thread* tersebut untuk memuat DLL yang diinjeksikan. Hal ini dilakukan dengan memaksa *thread* tersebut menjalankan fungsi `LoadLibrary`:

```
HINSTANCE LoadLibrary( PCTSTR pszLibFile );
```

Perhatikan definisi fungsi `LoadLibrary` dalam *file header* `WinBase.h` berikut:

```
HINSTANCE WINAPI LoadLibraryA( LPCSTR pszLibFileName );
HINSTANCE WINAPI LoadLibraryW( LPCWSTR pszLibFileName );
#ifdef UNICODE
#define LoadLibrary LoadLibraryW
#else
#define LoadLibrary LoadLibraryA
#endif // !UNICODE
```

Sebenarnya, ada dua fungsi `LoadLibrary` yang ada, yaitu `LoadLibraryA` dan `LoadLibraryW`. Perbedaan antara keduanya ada pada tipe parameter yang dilewatkan ke fungsi. Jika penulis memiliki nama *file* pustaka yang disimpan sebagai *string* ANSI, penulis harus memanggil `LoadLibraryA` (A berarti ANSI). Sementara itu, jika penulis memiliki nama *file* pustaka yang disimpan sebagai *string* Unicode, penulis harus memanggil `LoadLibraryW` (W berarti karakter lebar atau *wide*). Jadi, sebenarnya tidak ada fungsi yang bernama `LoadLibrary`, yang ada hanyalah `LoadLibraryA` dan `LoadLibraryW`. Untuk sebagian besar aplikasi, `macro` `LoadLibrary` memanggil fungsi `LoadLibraryA`.

Untungnya, prototipe fungsi `LoadLibrary` sama dengan prototipe fungsi *thread*. Berikut ini merupakan prototipe fungsi *thread*:

```
DWORD WINAPI ThreadFunc( PVOID pvParam );
```

Meskipun prototipe kedua fungsi tidak sama persis, prototipe kedua fungsi tersebut cukup mirip untuk dapat saling dipertukarkan. Kedua fungsi menerima sebuah parameter dan keduanya mengembalikan sebuah nilai. Selain itu, keduanya juga menggunakan konvensi pemanggilan yang sama, yaitu `WINAPI`. Ini sangat menguntungkan karena berarti penulis hanya perlu menciptakan *thread* baru dengan alamat `LoadLibraryA` atau `LoadLibraryW` sebagai alamat fungsi *thread*. Pada dasarnya, yang perlu dilakukan penulis hanyalah menjalankan kode seperti berikut:

```
HANDLE hThread = CreateRemoteThread( hProcessRemote, NULL, 0,
    LoadLibraryA, "C:\\MyLib.dll", 0, NULL );
```

Atau, bila penulis ingin memanggil versi Unicode-nya, seperti berikut:

```
HANDLE hThread = CreateRemoteThread( hProcessRemote, NULL, 0,
    LoadLibraryW, L"C:\\MyLib.dll", 0, NULL );
```

Saat *thread* baru tercipta dalam proses yang ingin diinjeksi, *thread* tersebut segera memanggil fungsi `LoadLibraryA` (atau `LoadLibraryW`) dengan melewati alamat *path* DLL yang akan diinjeksikan. Meskipun demikian, sebelum DLL tersebut dapat diinjeksikan, ada dua masalah yang harus diselesaikan.

Masalah pertama adalah bahwa penulis tidak bisa serta-merta melewati `LoadLibraryA` atau `LoadLibraryW` sebagai parameter keempat `CreateRemoteThread`, seperti terlihat di atas. Ketika penulis meng-*compile* dan me-*link* sebuah program, program yang dihasilkan memiliki *import section*. *Import section* ini terdiri atas sekumpulan *thunk* ke fungsi-fungsi yang diimpor. Ketika penulis memanggil fungsi seperti `LoadLibraryA`, fungsi yang dipanggil sebenarnya bukanlah fungsi yang bersangkutan, melainkan *thunk* yang kemudian melompat ke fungsi yang bersangkutan.

Jika penulis menggunakan *pointer* langsung ke `LoadLibraryA` dalam panggilan `CreateRemoteThread`, *pointer* tersebut merujuk ke alamat *thunk* `LoadLibraryA` dalam *import section* modul penulis. Melewatkan alamat *thunk* sebagai alamat awal *thread* baru menyebabkan *thread* baru tersebut untuk menjalankan kode yang tidak dapat diprediksi, dan kemungkinan menyebabkan pelanggaran akses. Untuk memaksa pemanggilan fungsi `LoadLibraryA` secara langsung (tanpa melalui *thunk*), penulis harus memperoleh alamat memori `LoadLibraryA` dengan memanggil `GetProcAddress`.

Panggilan `CreateRemoteThread` mengasumsikan bahwa `Kernel32.dll` dipetakan pada lokasi memori yang sama dalam ruang alamat proses penulis dan proses yang ingin diinjeksi. Setiap aplikasi memerlukan `Kernel32.dll`, dan

biasanya sistem memetakan Kernel32.dll pada alamat yang sama dalam setiap proses. Jadi, penulis harus memanggil `CreateRemoteThread` seperti berikut:

```
// Peroleh alamat LoadLibraryA yang sebenarnya (dalam Kernel32.dll)
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
    GetProcAddress( GetModuleHandle( TEXT( "Kernel32" ) ),
        "LoadLibraryA" );

HANDLE hThread = CreateRemoteThread( hProcessRemote, NULL, 0,
    pfnThreadRtn, "C:\\MyLib.dll", 0, NULL );
```

Atau, bila penulis ingin memanggil versi Unicode-nya, seperti berikut:

```
// Peroleh alamat LoadLibraryW yang sebenarnya (dalam Kernel32.dll)
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
    GetProcAddress( GetModuleHandle( TEXT( "Kernel32" ) ),
        "LoadLibraryW" );

HANDLE hThread = CreateRemoteThread( hProcessRemote, NULL, 0,
    pfnThreadRtn, L"C:\\MyLib.dll", 0, NULL );
```

Sementara itu, masalah kedua berkaitan dengan *string path* DLL. *String path* DLL "C:\\MyLib.dll" ada dalam ruang alamat proses pemanggil. Alamat *string* ini diberikan ke *thread* baru, yang langsung dilewatkan ke `LoadLibraryA`. Akan tetapi, saat `LoadLibraryA` mencoba mengakses alamat memori tersebut, *string path* DLL yang dimaksud tidak ada di situ dan *thread* tersebut kemungkinan akan mengalami pelanggaran akses.

Untuk mengatasi masalah ini, penulis perlu membawa *string path* DLL tersebut ke ruang alamat proses yang ingin diinjeksi. Setelah itu, saat memanggil `CreateRemoteThread`, penulis perlu melewatkan alamat *string* tersebut (dalam ruang alamat proses yang ingin diinjeksi). Sekali lagi, Windows memiliki fungsi `VirtualAllocEx` dan `VirtualFreeEx` yang memungkinkan sebuah proses untuk mengalokasikan (dan membebaskan) memori dalam ruang alamat proses lain:

```
PVOID VirtualAllocEx(
    HANDLE hProcess,
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect );
```

```
BOOL VirtualFreeEx(
    HANDLE hProcess,
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD dwFreeType );
```

Kedua fungsi tersebut mirip dengan versi "tanpa Ex"-nya. Perbedaannya adalah kedua fungsi tersebut memerlukan *handle* proses sebagai argumen pertamanya. *Handle* ini menentukan di proses mana tindakan ingin dilakukan.

Setelah penulis mengalokasikan memori untuk *string*, penulis juga memerlukan cara untuk menyalin *string* dari ruang alamat proses penulis ke ruang alamat proses yang ingin diinjeksi. Windows memiliki fungsi yang memungkinkan proses untuk membaca dan menulis ruang alamat proses lain:

```
BOOL ReadProcessMemory(
    HANDLE hProcess,
    PVOID pvAddressRemote,
    PVOID pvBufferLocal,
    DWORD dwSize,
    PDWORD pdwNumBytesRead );
```

```
BOOL WriteProcessMemory(
    HANDLE hProcess,
    PVOID pvAddressRemote,
    PVOID pvBufferLocal,
    DWORD dwSize,
    PDWORD pdwNumBytesWritten );
```

Proses yang ingin diinjeksi dikenali dengan parameter *hProcess*. Parameter *pvAddressRemote* menentukan alamat memori dalam ruang alamat proses yang ingin diinjeksi, *pvBufferLocal* adalah alamat memori dalam proses penulis, *dwSize* adalah banyaknya byte yang ingin disalin, dan *pdwNumBytesRead* serta *pdwNumBytesWritten* menentukan banyaknya byte yang disalin (kedua nilai tersebut dapat diperiksa setelah fungsi kembali).

Dengan demikian, dapat disimpulkan langkah-langkah yang harus dilakukan untuk melakukan *DLL injection* melalui *remote thread*:

1. Gunakan fungsi `VirtualAllocEx` untuk mengalokasikan memori dalam ruang alamat proses yang ingin diinjeksi.

2. Gunakan fungsi `WriteProcessMemory` untuk menyalin *path* DLL yang ingin diinjeksikan ke memori yang dialokasikan pada langkah 1.
3. Gunakan fungsi `GetProcAddress` untuk memperoleh alamat fungsi `LoadLibraryA` atau `LoadLibraryW` yang sebenarnya (dalam `Kernel32.dll`).
4. Gunakan fungsi `CreateRemoteThread` untuk menciptakan *thread* dalam proses yang ingin diinjeksi yang memanggil fungsi `LoadLibraryA` atau `LoadLibraryW`, dengan melewati alamat memori yang dialokasikan pada langkah 1.

Pada saat ini, DLL telah diinjeksikan ke ruang alamat proses yang ingin diinjeksi, dan fungsi `DllMain` DLL tersebut akan menerima pemberitahuan `DLL_PROCESS_ATTACH` dan dapat menjalankan kode yang diinginkan. Saat `DllMain` kembali, *thread* akan kembali ke `LoadLibrary`, lalu ke fungsi `BaseThreadStart`. `BaseThreadStart` kemudian akan memanggil `ExitThread`, menyebabkan *thread* berhenti.

Proses yang diinjeksi masih memiliki memori yang dialokasikan pada langkah 1 dan DLL yang diinjeksikan juga masih dipetakan dalam ruang alamatnya. Untuk membersihkan hal-hal tersebut, hal-hal berikut harus dilakukan setelah *thread* selesai:

5. Gunakan fungsi `VirtualFreeEx` untuk membebaskan memori yang dialokasikan pada langkah 1.
6. Gunakan fungsi `GetProcAddress` untuk memperoleh alamat fungsi `FreeLibrary` yang sebenarnya (dalam `Kernel32.dll`).
7. Gunakan fungsi `CreateRemoteThread` untuk menciptakan *thread* dalam proses yang diinjeksi yang memanggil fungsi `FreeLibrary`, dengan melewati `HINSTANCE` DLL yang diinjeksikan.

II.2.5. DLL Injection dengan DLL Trojan

Sebuah cara lain untuk melakukan *DLL injection* adalah dengan mengganti DLL yang diketahui akan dimuat proses. Sebagai contoh, jika penulis mengetahui bahwa sebuah proses akan memuat *XYZ.dll*, penulis dapat menciptakan DLL dengan nama yang sama. Tentu saja, *XYZ.dll* yang asli harus diubah namanya.

Dalam *XYZ.dll* penulis, penulis harus mengekspor semua simbol yang diekspor *XYZ.dll* yang asli. Hal ini dapat dilakukan dengan *function forwarder* (yang juga dapat meng-*hook* fungsi-fungsi tertentu), tetapi teknik ini seharusnya dihindari karena tidak tahan perubahan versi. Sebagai contoh, jika penulis mengganti sebuah DLL sistem dan di kemudian hari Microsoft menambahkan fungsi-fungsi baru dalam DLL tersebut, DLL penulis tidak akan memiliki *function forwarder* untuk fungsi-fungsi baru tersebut. Akibatnya, aplikasi-aplikasi yang menggunakan fungsi-fungsi baru tersebut tidak akan dapat berjalan.

Jika penulis hanya ingin menginjeksi sebuah aplikasi dengan teknik ini, penulis bisa saja menciptakan DLL baru dengan nama yang berbeda dan mengubah *import section* modul EXE aplikasi tersebut. Ini dapat dilakukan karena *import section* suatu modul mengandung nama-nama DLL yang diperlukan modul tersebut. Jika penulis menambahkan sebuah DLL ke *import section* suatu modul, DLL tersebut akan dimuat setiap kali modul berjalan.

II.2.6. DLL Injection Sebagai Debugger

Sebuah *debugger* dapat melakukan tindakan-tindakan khusus pada proses yang di-*debug* (*debuggee*). Ketika proses yang di-*debug* dimuat, sistem secara otomatis akan memberitahu *debugger* saat ruang alamat proses yang di-*debug* telah siap, tetapi sebelum *thread* utama proses yang

di-*debug* berjalan. Pada saat itu, *debugger* dapat menuliskan kode dalam ruang alamat proses yang di-*debug* (misalnya dengan `WriteProcessMemory`) lalu memaksa *thread* utama proses yang di-*debug* untuk menjalankan kode tersebut.

Teknik ini mengharuskan penulis memanipulasi struktur `CONTEXT` *thread* yang di-*debug*, berarti penulis harus menulis kode khusus untuk CPU tertentu (*CPU-specific code*). Akibatnya, penulis harus memodifikasi program penulis agar dapat bekerja di berbagai *platform* CPU. Selain itu, penulis mungkin harus menulis instruksi mesin yang akan dijalankan proses yang di-*debug* secara manual. Akhirnya, hubungan antara *debugger* dengan proses yang di-*debug* adalah permanen (apabila *debugger*-nya berhenti berjalan, Windows secara otomatis akan menghentikan proses yang di-*debug*).

II.2.7. DLL Injection dengan `CreateProcess`

Apabila proses penulis merupakan proses yang menjalankan proses yang ingin diinjeksi, *DLL injection* dapat dilakukan dengan lebih mudah. Ini karena proses penulis (proses induk) dapat menjalankan proses yang ingin diinjeksi (proses anakan) dalam keadaan berhenti (*suspended*). Cara ini memungkinkan penulis untuk mengubah proses anakan tanpa mempengaruhi eksekusinya, karena proses anakan memang belum mulai berjalan. Selain itu, proses induk juga memperoleh *handle thread* utama proses anakan. Dengan *handle* ini, penulis dapat mengubah kode yang akan dieksekusi *thread* tersebut.

Berikut ini merupakan salah satu cara proses penulis untuk mengendalikan kode yang dieksekusi *thread* utama proses anakan:

1. Jalankan proses anakan dalam keadaan berhenti.

2. Dapatkan alamat memori awal *thread* utama proses anakan dari *header file* EXE-nya.
3. Simpan instruksi-instruksi mesin pada alamat memori ini.
4. Tuliskan beberapa instruksi mesin pada alamat ini. Instruksi-instruksi tersebut harus memanggil `LoadLibrary` untuk memuat DLL penulis.
5. Lanjutkan *thread* utama proses anakan sehingga kode tersebut dijalankan.
6. Kembalikan instruksi-instruksi asli pada alamat awal tersebut.
7. Biarkan proses anakan kembali berjalan dari alamat awal, seolah-olah tidak terjadi sesuatu.

Teknik ini memiliki banyak keunggulan. Pertama, teknik ini memungkinkan perubahan ruang alamat aplikasi sebelum aplikasi berjalan. Kedua, karena proses penulis bukan *debugger*, penulis dapat men-*debug* aplikasi dengan DLL yang diinjeksikan. Terakhir, teknik ini dapat bekerja pada aplikasi *console* dan GUI.

Tentu saja, teknik ini juga memiliki kelemahan. Pertama, penulis hanya dapat menginjeksikan DLL apabila proses penulis adalah proses induk. Kedua, teknik ini terikat CPU (*CPU-dependent*): program penulis harus disesuaikan untuk *platform* CPU yang berbeda-beda.

II.2.8. API Hooking

Menginjeksikan DLL ke ruang alamat proses merupakan cara yang bagus untuk mengetahui apa saja yang terjadi dalam sebuah proses. Akan tetapi, kadang menginjeksikan DLL saja belum memberikan informasi yang cukup. Kadang penulis ingin mengetahui bagaimana *thread-thread* dalam sebuah proses memanggil berbagai fungsi, dan kadang

penulis juga ingin memodifikasi hal-hal yang dilakukan sebuah fungsi Windows.

Sebagai contoh, andaikan penulis menulis DLL untuk sebuah aplikasi. DLL tersebut berfungsi meningkatkan fungsionalitas aplikasi (semacam *plug-in*). Saat aplikasi berhenti berjalan, DLL penulis akan menerima pemberitahuan `DLL_PROCESS_DETACH` dan akan menjalankan kode pembersihnya (*cleanup code*). DLL penulis akan memanggil fungsi-fungsi dalam DLL-DLL lain untuk menutup koneksi-koneksi, *file-file*, dan objek-objek lainnya. Akan tetapi, saat DLL penulis menerima pemberitahuan `DLL_PROCESS_DETACH`, banyak DLL lain dalam ruang alamat aplikasi yang telah menerima pemberitahuan `DLL_PROCESS_DETACH`. Akibatnya, saat DLL penulis mencoba membersihkan dirinya, banyak fungsi-fungsi yang dipanggilnya yang gagal karena telah banyak DLL lain yang tidak lagi dimuat (*unloaded*).

Untuk menyelesaikan masalah di atas dengan *hooking*, penulis *meng-hook* fungsi `ExitProcess`. Sebagaimana diketahui, pemanggilan `ExitProcess` akan menyebabkan sistem memberitahu DLL-DLL yang dimuat dengan pemberitahuan `DLL_PROCESS_DETACH`. Dengan *meng-hook* fungsi `ExitProcess`, penulis memastikan bahwa DLL penulis akan diberitahu saat `ExitProcess` dipanggil. Pemberitahuan ini akan datang sebelum DLL-DLL lain menerima pemberitahuan `DLL_PROCESS_DETACH`, sehingga DLL-DLL lain dalam aplikasi masih dimuat dan bekerja dengan baik. Pada saat itu, DLL penulis akan mengetahui bahwa aplikasi akan segera berhenti berjalan, sehingga menjalankan kode pembersihnya. Setelah itu, fungsi `ExitProcess` sistem operasi akan dipanggil, menyebabkan semua DLL menerima pemberitahuan `DLL_PROCESS_DETACH` dan membersihkan dirinya masing-masing. DLL penulis sendiri tidak perlu melakukan

pembersihan khusus saat menerima pemberitahuan tersebut karena DLL penulis sebelumnya telah membersihkan dirinya.

Dalam contoh ini, penginjeksian DLL dapat dilakukan dengan mudah, karena aplikasi memang telah dirancang untuk melakukan hal itu. Saat DLL penulis dimuat, DLL penulis harus mencari panggilan-panggilan ke `ExitProcess` dalam semua modul yang dimuat. Ketika DLL penulis menemukan panggilan ke `ExitProcess`, DLL penulis harus memodifikasi modul tersebut agar modul tersebut memanggil fungsi pengganti penulis, bukan fungsi `ExitProcess` sistem operasi. Setelah fungsi pengganti penulis (fungsi *hook*) menjalankan kode pembersihnya, barulah fungsi `ExitProcess` sistem operasi (dalam `Kernel32.dll`) akan dipanggil.

II.2.8.1. API Hooking dengan Penimpaan Kode

API hooking bukanlah barang baru – para pengembang telah menggunakan berbagai metode *API hooking* selama bertahun-tahun. Salah satu metode *API hooking* paling awal yang digunakan adalah dengan penimpaan kode (*code overwriting*), sebagai berikut:

1. Penulis mencari alamat fungsi yang ingin di-*hook* dalam memori (misalnya, fungsi `ExitProcess` dalam `Kernel32.dll`).
2. Penulis menyimpan beberapa byte pertama fungsi ini di tempat terpisah.
3. Penulis menimpa beberapa byte pertama fungsi ini dengan instruksi CPU `JUMP` yang melompat ke alamat memori fungsi pengganti. Tentu saja, fungsi pengganti harus memiliki *signature* yang sama dengan fungsi yang di-*hook*: semua parameternya harus sama, kembaliannya harus sama, dan konvensi pemanggilannya juga harus sama.

4. Sekarang, ketika sebuah *thread* memanggil fungsi yang di-*hook*, instruksi JUMP akan melompat ke fungsi pengganti. Pada saat itu, penulis dapat menjalankan kode apapun yang dikehendaki.
5. Penulis tidak lagi meng-*hook* (*unhook*) fungsi tersebut dengan mengembalikan byte-byte yang disimpan (pada langkah 2) pada awal fungsi yang di-*hook*.
6. Penulis memanggil fungsi yang di-*hook* (yang sedang tidak di-*hook*), sehingga fungsi tersebut berjalan seperti biasa.
7. Saat fungsi yang asli kembali, penulis kembali menjalankan langkah 2 dan 3 di atas sehingga fungsi pengganti akan dijalankan di waktu berikutnya.

Metode ini banyak digunakan para pemrogram Windows 16-bit dan bekerja dengan baik pada lingkungan itu. Meskipun demikian, metode ini kini memiliki banyak keterbatasan dan seharusnya tidak lagi digunakan. Pertama-tama, metode ini terikat CPU: instruksi JUMP pada CPU x86, Alpha, dan CPU-CPU lainnya berbeda, dan program juga harus menggunakan instruksi mesin yang ditulis secara manual. Kedua, metode ini tidak akan bekerja dalam lingkungan *preemptive multithreading*. Hal ini karena penggantian kode pada awal fungsi memakan waktu. Sementara itu, saat kode sedang diganti, *thread* lain mungkin memanggil fungsi tersebut (dengan hasil yang tidak dapat diprediksi). Oleh karena itu, metode ini hanya dapat bekerja jika tidak lebih dari satu *thread* memanggil fungsi yang di-*hook* pada suatu saat.

II.2.8.2. API Hooking dengan Perubahan Import Section Modul

Ternyata, ada teknik *API hooking* yang dapat menyelesaikan kedua masalah di atas. Sebagaimana diketahui, *import section* suatu modul mengandung kumpulan DLL yang diperlukan modul tersebut agar dapat berjalan. Selain itu, *import section* suatu modul juga mengandung daftar simbol yang diimpor modul tersebut dari setiap DLL. Ketika suatu modul memanggil fungsi yang diimpor, modul tersebut sebenarnya memperoleh alamat fungsi yang diimpor dari *import section*, lalu melompat ke alamat tersebut. Jadi, untuk meng-*hook* sebuah fungsi, yang perlu dilakukan hanyalah mengubah alamatnya dalam *import section* modul.

Fungsi berikut ini melakukan hal tersebut. Fungsi tersebut mencari rujukan ke simbol pada alamat tertentu dalam *import section* suatu modul. Jika ada rujukan semacam itu, fungsi tersebut akan mengganti alamat simbolnya.

```
void ReplaceIATEntryInOneMod( PCSTR pszCalleeModName,
    PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller ) {

    ULONG ulSize;
    PIMAGE_IMPORT_DESCRIPTOR pImportDesc =
        (PIMAGE_IMPORT_DESCRIPTOR) ImageDirectoryEntryToData(
            hmodCaller, TRUE, IMAGE_DIRECTORY_ENTRY_IMPORT,
            &ulSize );

    if( pImportDesc == NULL )
        return; // Modul ini tidak memiliki import section.

    // Cari import descriptor yang berisi rujukan ke modul yang
    // mengandung fungsi yang ingin di-hook.
    for( ; pImportDesc->Name; pImportDesc++ ) {
        PSTR pszModName = (PSTR)( (PBYTE)hmodCaller +
            pImportDesc->Name );
        if( lstrcmpiA( pszModName, pszCalleeModName ) == 0 )
            break;
    }

    if( pImportDesc->Name == 0 )
        // Modul ini tidak mengimpor fungsi dari modul yang
        // mengandung fungsi yang ingin di-hook.
```

```

return;

// Peroleh import address table (IAT) modul yang mengandung
// fungsi yang ingin di-hook.
PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
    ( (PBYTE)hmodCaller + pImportDesc->FirstThunk );

// Ganti alamat fungsi yang ingin di-hook dengan alamat fungsi
// penggantinya.
for( ; pThunk->ul.Function; pThunk++ ) {

    // Peroleh alamat fungsi tersebut.
    PROC* ppfn = (PROC*) &pThunk->ul.Function;

    // Apakah fungsi ini adalah fungsi yang dicari?
    BOOL fFound = ( *ppfn == pfnCurrent );

    if( fFound ) {
        // Alamatnya cocok, ubah alamat dalam import section.
        WriteProcessMemory( GetCurrentProcess(), ppfn,
            &pfnNew, sizeof(pfnNew), NULL );
        return; // Selesai.
    }
}

// Jika fungsi sampai ke sini, berarti fungsi yang ingin di-
// hook tidak ada dalam import section modul.
}

```

Untuk mengetahui cara memanggil fungsi ini, penulis akan memulai dengan menjelaskan sebuah lingkungan. Andaikan penulis memiliki modul bernama Aplikasi.exe. Kode modul tersebut memanggil fungsi ExitProcess dalam Kernel32.dll, tetapi penulis ingin agar modul tersebut memanggil fungsi MyExitProcess dalam Plugin.dll penulis. Untuk melakukan hal ini, penulis memanggil ReplaceIATEntryInOneMod sebagai berikut:

```

PROC pfnOrig = GetProcAddress( GetModuleHandle( "Kernel32" ),
    "ExitProcess" );
HMODULE hmodCaller = GetModuleHandle( "Aplikasi.exe" );

void ReplaceIATEntryInOneMod(
    "Kernel32.dll", // Modul yang mengandung fungsi yang ingin
                  // di-hook
    pfnOrig,       // Alamat fungsi yang ingin di-hook
    MyExitProcess, // Alamat fungsi pengganti
    hmodCaller ); // Handle modul yang memanggil fungsi yang
                  // ingin di-hook

```

Hal pertama yang dilakukan `ReplaceIATEntryInOneMod` adalah mencari `import section` modul `hmodCaller` dengan memanggil `ImageDirectoryEntryToData`, dengan melewati `IMAGE_DIRECTORY_ENTRY_IMPORT` kepadanya. Jika fungsi ini menghasilkan `NULL`, berarti modul `Aplikasi.exe` tidak memiliki `import section` sehingga tidak ada yang perlu dilakukan.

Jika modul `Aplikasi.exe` memiliki `import section`, `ImageDirectoryEntryToData` akan mengembalikan alamat `import section` tersebut dalam bentuk `pointer` bertipe `PIMAGE_IMPORT_DESCRIPTOR`. Penulis kemudian harus mencari DLL yang berisi fungsi yang ingin di-hook dalam `import section` tersebut. Dalam contoh ini, penulis mencari simbol-simbol yang diimpor dari `Kernel32.dll` (parameter pertama yang dilewatkan ke fungsi `ReplaceIATEntryInOneMod`). Sebuah `loop for` kemudian mencari nama modul DLL yang dimaksud. Perhatikan bahwa semua `string` dalam `import section` modul disimpan sebagai `string ANSI` (bukan `Unicode`). Itulah mengapa penulis langsung memanggil fungsi `lstrcmpiA`, bukan `macro lstrcmpi`.

Jika `loop` berhenti tanpa menemukan rujukan ke simbol-simbol dalam `Kernel32.dll`, fungsi akan kembali tanpa melakukan apapun. Jika `import section` modul merujuk ke simbol-simbol dalam `Kernel32.dll`, penulis memperoleh alamat larik struktur `IMAGE_THUNK_DATA` yang berisi informasi simbol-simbol yang diimpor. Setelah itu, penulis menelusuri simbol-simbol yang diimpor dari `Kernel32.dll`, mencari alamat yang sama dengan alamat simbol sekarang. Dalam contoh ini, penulis mencari alamat yang sama dengan alamat fungsi `ExitProcess`.

Jika tidak ada alamat yang sama dengan alamat yang dicari, modul tersebut pasti tidak mengimpor simbol yang

dicari, dan `ReplaceIATEntryInOneMod` akan kembali. Jika alamat yang dicari ditemukan, `WriteProcessMemory` akan dipanggil untuk mengubah alamat fungsi pengganti. Fungsi `WriteProcessMemory` digunakan karena `WriteProcessMemory` mengubah byte-byte memori tanpa memedulikan proteksi halaman byte-byte tersebut.

Sejak saat ini, ketika sebuah *thread* menjalankan kode dalam modul `Aplikasi.exe` yang memanggil `ExitProcess`, *thread* tersebut akan memanggil fungsi pengganti penulis. Dari fungsi tersebut, penulis dapat memperoleh alamat fungsi `ExitProcess` yang sebenarnya dan memanggilnya saat penulis menginginkan pemrosesan `ExitProcess` secara normal.

Perhatikan bahwa fungsi `ReplaceIATEntryInOneMod` mengubah panggilan-panggilan fungsi dari kode dalam sebuah modul. Akan tetapi, sebuah DLL lain dalam ruang alamat mungkin memanggil `ExitProcess` juga. Jika sebuah modul lain selain `Aplikasi.exe` memanggil `ExitProcess`, panggilannya akan memanggil fungsi `ExitProcess` dalam `Kernel32.dll`.

Jika penulis ingin menjebak semua panggilan `ExitProcess` dari semua modul, penulis harus memanggil `ReplaceIATEntryInOneMod` untuk setiap modul yang dimuat dalam ruang alamat proses. Untuk melakukan hal itu, penulis telah menulis fungsi lain bernama `ReplaceIATEntryInAllMods`. Fungsi ini menggunakan fungsi `toolhelp` untuk memperoleh semua modul yang dimuat dalam ruang alamat proses dan memanggil `ReplaceIATEntryInOneMod` untuk setiap modul, dengan melewati *handle* modul sebagai parameter terakhir.

Beberapa masalah masih dapat terjadi. Sebagai contoh, bagaimana jika sebuah *thread* memanggil `LoadLibrary` untuk memuat sebuah DLL baru setelah penulis

memanggil `ReplaceIATEntryInAllMods`? Dalam keadaan ini, DLL yang baru dimuat mungkin memanggil `ExitProcess` yang belum di-hook. Untuk menyelesaikan masalah ini, penulis harus meng-hook fungsi `LoadLibraryA`, `LoadLibraryW`, `LoadLibraryExA`, dan `LoadLibraryExW` agar penulis dapat menjebak panggilan-panggilan tersebut dan memanggil `ReplaceIATEntryInOneMod` untuk modul yang baru dimuat.

Masalah terakhir berkaitan dengan `GetProcAddress`. Andaikan sebuah *thread* menjalankan kode berikut:

```
typedef int (WINAPI *PFNEXITPROCESS)( UINT uExitCode );
PFNEXITPROCESS pfnExitProcess = (PFNEXITPROCESS)GetProcAddress(
    GetModuleHandle( "Kernel32" ), "ExitProcess" );
pfnExitProcess( 0 );
```

Kode di atas memerintahkan sistem untuk memperoleh alamat `ExitProcess` yang sebenarnya (dalam `Kernel32.dll`), lalu memanggil alamat tersebut. Jika sebuah *thread* menjalankan kode tersebut, fungsi pengganti penulis tidak akan dipanggil. Untuk mengatasi masalah ini, penulis juga harus meng-hook fungsi `GetProcAddress`. Jika fungsi tersebut dipanggil dan akan mengembalikan alamat fungsi yang di-hook, penulis akan mengembalikan alamat fungsi penggantinya.

II.3. Virtualisasi

II.3.1. Pengantar Virtualisasi

Pada hakekatnya, virtualisasi memungkinkan transformasi perangkat keras menjadi perangkat lunak. Dengan aplikasi virtualisasi, sumber daya perangkat keras sebuah komputer, termasuk CPU, RAM, *hard disk*, dan *network card*-nya, dapat ditransformasikan ("divirtualisasikan") untuk menciptakan *virtual machine* yang dapat menjalankan sistem operasi dan aplikasi-aplikasinya sendiri, seperti komputer "sungguhan".

Berbagai *virtual machine* berbagi sumber daya perangkat keras tanpa mengganggu sesamanya, sehingga berbagai sistem operasi dan aplikasi dapat dijalankan dengan aman pada saat yang sama dalam sebuah komputer.

Cara kerja virtualisasi adalah dengan "menyisipkan" lapisan perangkat lunak langsung di atas perangkat keras atau di sistem operasi komputer *host*. Lapisan perangkat lunak ini menciptakan *virtual machine* dan mengandung *virtual machine monitor* atau *hypervisor* yang mengalokasikan sumber daya perangkat keras secara dinamis dan transparan, sehingga berbagai sistem operasi dapat berjalan bersama-sama dalam sebuah komputer tanpa mengetahuinya.

II.3.2. Sejarah Virtualisasi

Virtualisasi pertama kali diimplementasikan lebih dari 30 tahun yang lalu oleh IBM sebagai cara mempartisi *mainframe*-nya menjadi *virtual machine*. Partisi-partisi tersebut memungkinkan *mainframe* untuk melakukan *multitasking*, yaitu menjalankan banyak aplikasi dan proses secara bersamaan. Karena pada waktu itu *mainframe* merupakan sumber daya yang mahal, mereka dirancang untuk dapat dipartisi untuk memaksimalkan nilai investasi.

Virtualisasi praktis ditinggalkan pada tahun 80-an dan 90-an, ketika aplikasi-aplikasi *client-server* dan *server-server* serta *desktop-desktop* x86 yang murah menetapkan model komputasi terdistribusi. Daripada membagi sumber daya secara terpusat dalam *mainframe*, perusahaan-perusahaan menggunakan sistem-sistem terdistribusi yang murah untuk membangun "pulau-pulau" (*islands*) kapasitas komputasi. Penggunaan Windows yang meluas dan kemunculan Linux sebagai sistem operasi server pada tahun 90-an menetapkan *server-server* x86 sebagai

standar industri. Perkembangan instalasi server dan desktop x86 memunculkan tantangan infrastruktur dan operasional IT baru, termasuk:

- a. **Rendahnya utilisasi infrastruktur:** Menurut International Data Corporation (IDC), sebuah lembaga riset, instalasi server x86 biasanya hanya memiliki utilisasi rata-rata 10% hingga 15% dari total kapasitasnya. Perusahaan-perusahaan biasanya hanya menjalankan sebuah aplikasi dalam sebuah server untuk menghindari resiko celah keamanan dalam sebuah aplikasi yang dapat mempengaruhi ketersediaan aplikasi-aplikasi lainnya dalam server.
- b. **Meningkatnya biaya infrastruktur fisik:** Biaya-biaya operasional untuk mendukung perkembangan infrastruktur fisik terus naik. Kebanyakan infrastruktur komputasi harus beroperasi terus-menerus, menghasilkan biaya listrik, pendingin, dan fasilitas yang terus dibebankan, berapapun utilisasi infrastrukturnya.
- c. **Meningkatnya biaya manajemen IT:** Dengan makin kompleksnya sistem-sistem komputer, tingkat pendidikan dan pengalaman yang diperlukan para pengelola infrastruktur, berikut biaya-biaya lainnya, pun terus meningkat. Perusahaan-perusahaan menghabiskan banyak waktu dan sumber daya untuk tugas-tugas manual perawatan server, sehingga membutuhkan lebih banyak orang untuk menyelesaikannya.
- d. **Kurangnya perlindungan terhadap bencana dan failover:** Semakin lama, perusahaan-perusahaan semakin terpengaruh *downtime server-server* dan *desktop-desktop-nya*. Ancaman serangan *hacker*,

bencana alam, wabah, dan terorisme telah meningkatkan pentingnya *contingency plan*, baik untuk *desktop* maupun *server*.

- e. **Perawatan *desktop***: Mengelola dan mengamankan *desktop-desktop* di lingkungan *enterprise* menghadirkan banyak tantangan. Mengendalikan *desktop-desktop* yang tersebar di berbagai tempat dan mengatur kebijakan pengelolaan, akses, dan keamanannya tanpa mengurangi efektivitas kerja para penggunanya adalah kompleks dan mahal. *Patch-patch* dan *upgrade-upgrade* harus terus diinstallkan ke *desktop-desktop* untuk menutup celah keamanan.

Pada tahun 1999, VMware memperkenalkan virtualisasi sistem x86 sebagai cara efisien mengatasi tantangan-tantangan di atas, dan untuk mentransformasikan sistem x86 menjadi infrastruktur yang menawarkan isolasi penuh, mobilitas, dan pilihan sistem operasi untuk lingkungan-lingkungan aplikasi.

Berbeda dengan *mainframe*, sistem x86 tidak dirancang untuk mendukung virtualisasi penuh, dan VMware harus mengatasi tantangan-tantangan besar untuk menciptakan *virtual machine* dalam komputer x86.

Fungsi dasar kebanyakan CPU, baik dalam *mainframe* maupun PC, adalah menjalankan deretan instruksi yang tersimpan (program). Dalam prosesor x86, ada 17 instruksi yang bermasalah ketika divirtualisasikan, menyebabkan sistem operasi untuk menampilkan peringatan, menghentikan aplikasi, atau bahkan *crash*. Akibatnya, ketujuhbelas instruksi tersebut menjadi tantangan besar implementasi awal virtualisasi dalam komputer x86.

Untuk menangani instruksi-instruksi bermasalah tersebut, VMware mengembangkan teknik virtualisasi

adaptif yang "menjebak" instruksi-instruksi tersebut ketika dibangkitkan dan mengubahnya menjadi instruksi-instruksi aman yang dapat divirtualisasikan, sementara mengizinkan instruksi-instruksi lainnya untuk langsung dijalankan. Hasilnya adalah *virtual machine* berperforma tinggi yang cocok dengan perangkat keras komputer fisik dan benar-benar kompatibel dengan semua perangkat lunak.

II.3.3. Manfaat Virtualisasi

- a. **Konsolidasi server dan optimisasi infrastruktur:** Virtualisasi memungkinkan utilisasi sumber daya yang jauh lebih tinggi dengan mengumpulkan sumber daya infrastruktur dan menghancurkan model "satu aplikasi per server".
- b. **Pengurangan biaya infrastruktur fisik:** Dengan virtualisasi, jumlah server dan perangkat-perangkat keras lainnya dapat dikurangi. Ini menyebabkan pengurangan penggunaan tempat, listrik, dan pendingin, menyebabkan pengurangan biaya IT.
- c. **Fleksibilitas dan ketanggapan operasional yang lebih baik:** Virtualisasi menawarkan cara baru mengelola infrastruktur IT, dan dapat membantu para administrator IT menghabiskan lebih sedikit waktu dalam tugas-tugas manual seperti instalasi, konfigurasi, pengawasan, dan perawatan.
- d. **Meningkatnya ketersediaan aplikasi dan keberlanjutan usaha:** Virtualisasi memungkinkan eliminasi *downtime* terencana dan pemulihan dari *outage* dengan cepat, dengan kemampuan *backup* dan migrasi keseluruhan infrastruktur *virtual* dengan aman tanpa penghentian layanan.
- e. **Pengelolaan dan keamanan desktop yang lebih baik:** Virtualisasi memungkinkan instalasi, pengelolaan, dan

pengawasan *desktop* yang dapat diakses pengguna, dari tempat yang sama (*locally*) atau tempat yang jauh (*remotely*), dengan atau tanpa sambungan jaringan, dari hampir semua *desktop*, *laptop*, atau *tablet PC* standar.

II.3.4. Pengantar *Virtual Machine*

Sebuah *virtual machine* merupakan "blok" perangkat lunak terisolasi yang dapat menjalankan sistem operasi dan aplikasi-aplikasinya sendiri, seperti komputer fisik. Sebuah *virtual machine* berperilaku persis seperti komputer fisik dan memiliki CPU, RAM, *hard disk*, dan *network card virtual*-nya sendiri.

Sebuah sistem operasi tidak dapat membedakan antara *virtual machine* dengan komputer fisik, demikian juga dengan aplikasi-aplikasi dan komputer-komputer lainnya dalam jaringan. Bahkan, *virtual machine* pun berpikir bahwa dirinya adalah komputer "sungguhan". Meskipun demikian, *virtual machine* tersusun sepenuhnya oleh perangkat lunak dan tidak memiliki komponen perangkat keras. Akibatnya, *virtual machine* menawarkan beberapa keunggulan terhadap komputer fisik.

II.3.5. Manfaat *Virtual Machine*

- a. **Kompatibilitas:** Seperti sebuah komputer fisik, sebuah *virtual machine* memiliki sistem operasi dan aplikasi-aplikasinya sendiri, serta memiliki semua komponen yang dimiliki komputer fisik (*motherboard*, *VGA card*, *network card*, dsb.) Akibatnya, *virtual machine* benar-benar kompatibel dengan semua sistem operasi, aplikasi, dan *driver x86* standar, sehingga *virtual machine* dapat digunakan untuk menjalankan semua aplikasi yang dapat dijalankan dalam komputer fisik x86.

- b. **Isolasi:** Meskipun *virtual machine* dapat berbagi sumber daya sebuah komputer fisik, mereka benar-benar terisolasi satu sama lain, seperti apabila mereka adalah komputer-komputer fisik yang terpisah. Jika ada empat *virtual machine* dalam sebuah komputer fisik dan salah satunya *crash*, ketiga *virtual machine* lainnya tetap tersedia. Isolasi adalah alasan penting mengapa ketersediaan dan keamanan aplikasi-aplikasi yang berjalan dalam lingkungan *virtual* jauh lebih tinggi daripada aplikasi-aplikasi yang berjalan dalam sistem tradisional tanpa virtualisasi.
- c. **Enkapsulasi:** Pada hakekatnya, sebuah *virtual machine* adalah sebuah "blok" perangkat lunak yang berisi ("mengkapsulasikan") sekumpulan sumber daya perangkat keras *virtual*, selain sistem operasi dan semua aplikasinya. Enkapsulasi menjadikan *virtual machine* sangat portabel dan mudah dikelola. Sebagai contoh, sebuah *virtual machine* dapat dipindahkan atau disalin dari satu tempat ke tempat lain seperti *file-file* lainnya, atau disimpan dalam media penyimpanan standar, dari *USB flash disk* sampai *Storage Area Network (SAN)* skala *enterprise*.
- d. **Kemerdekaan terhadap perangkat keras:** *Virtual machine* benar-benar merdeka dari perangkat keras fisik tempat mereka berjalan. Sebagai contoh, sebuah *virtual machine* dapat dikonfigurasi dengan komponen-komponen *virtual* (*CPU*, *network card*, *SCSI controller*, dsb.) yang benar-benar berbeda dengan komponen-komponen fisik yang ada dalam perangkat keras. *Virtual machine* dalam sebuah komputer fisik bahkan dapat menjalankan sistem operasi yang berbeda-beda (*Windows*, *Linux*, dsb.)

Bersama enkapsulasi dan kompatibilitas, kemerdekaan terhadap perangkat keras memungkinkan kebebasan perpindahan sebuah *virtual machine* dari komputer x86 satu ke yang lainnya tanpa perubahan *driver*, sistem operasi, maupun aplikasi. Kemerdekaan terhadap perangkat keras juga memungkinkan menjalankan berbagai sistem operasi dan aplikasi dalam sebuah komputer fisik.

II.4. Internet dan World Wide Web

II.4.1. Internet

Internet merupakan kumpulan jaringan dan *gateway* di seluruh dunia yang menggunakan protokol TCP/IP untuk berkomunikasi satu sama lain. Pada jantung Internet terdapat *backbone* berupa jalur komunikasi data berkecepatan tinggi antara simpul-simpul (*node*) atau komputer-komputer (*host*) utama yang terdiri atas ribuan sistem komputer komersial, pemerintahan, pendidikan, dan lainnya, yang menyampaikan data-data dan pesan-pesan. Satu atau lebih simpul Internet dapat mati tanpa membahayakan Internet secara keseluruhan atau menyebabkan komunikasi dalam Internet menjadi berhenti, karena tidak ada komputer atau jaringan yang mengendalikannya. Internet bermula dari jaringan terdesentralisasi bernama ARPANET yang diciptakan Departemen Pertahanan Amerika Serikat pada tahun 1969 untuk memfasilitas komunikasi saat terjadi serangan nuklir. Pada akhirnya, jaringan-jaringan lainnya, termasuk BITNET, Usenet, UUCP, dan NSFnet, tersambung ke ARPANET. Saat ini, Internet menawarkan sejumlah layanan kepada penggunanya, seperti FTP, *e-mail*, World Wide Web, Usenet news, Gopher, IRC, *telnet*, dan sebagainya.

II.4.2. World Wide Web

World Wide Web merupakan kumpulan dokumen hiperteks yang saling berkaitan dalam *server-server* HTTP di seluruh dunia. Dokumen-dokumen dalam World Wide Web, disebut halaman atau halaman web, ditulis dalam bahasa HTML (*Hypertext Markup Language*), dikenali dengan URL (*Uniform Resource Locator*) yang menetapkan dalam komputer dan *path* manakah sebuah *file* dapat diakses, ditransmisikan dari *server* ke pengguna lewat protokol HTTP (*Hypertext Transfer Protocol*). Kode-kode, disebut *tag*, dalam sebuah dokumen HTML menghubungkan kata-kata atau citra-citra tertentu dalam dokumen dengan URL, sehingga seorang pengguna dapat mengakses *file* lainnya, yang mungkin terletak di bagian dunia yang lain, dengan penekanan tombol atau klik *mouse*. *File-file* tersebut dapat berisi teks (dalam berbagai jenis huruf dan gaya), citra, video, dan suara, serta *applet* Java, *control* ActiveX, dan program-program kecil yang berjalan saat pengguna mengaktifkannya dengan mengklik sebuah *link*. Seorang pengguna yang mengunjungi sebuah halaman web juga dapat mengunduh (*download*) *file* dari situs FTP dan berkirim pesan ke pengguna lainnya lewat *e-mail* dengan menggunakan *link* pada halaman web. World Wide Web dikembangkan oleh Timothy Berners-Lee pada tahun 1989 untuk laboratorium fisika partikel Eropa (*Conseil Européen pour le Recherche Nucléaire* atau CERN).