

**INSTITUT TEKNOLOGI DEL**

# **PEMINDAIAN MEMORI PROSES UNTUK PENDETEKSIAN PIRANTI LUNAK JAHAT**

**TUGAS AKHIR**

**Diajukan sebagai salah satu syarat untuk memperoleh gelar Sarjana Teknik Program Studi Sarjana Sistem Informasi**

|  |  |
| --- | --- |
| 12S14009 | YUSUF PANGARIBUAN |
| 12S14042 | LASRIA OSELINA NAINGGOLAN |
| 12S14045 | CLARA SONYA ANGELICA ARITONANG |

**FAKULTAS TEKNIK INFORMATIKA DAN ELEKTRO**

**PROGRAM STUDI SARJANA SISTEM INFORMASI**

**LAGUBOTI**

**AGUSTUS 2018**

HALAMAN PERNYATAAN ORISINALITAS

Tugas Akhir ini adalah hasil karya kami sendiri,

dan semua sumber baik yang dikutip maupun dirujuk

telah kami nyatakan dengan benar.

Nama : Yusuf Pangaribuan

NIM : 12S14009

Tanda Tangan :

Tanggal : 20 Agustus 2018

Nama : Lasria oselina Nainggolan

NIM : 12S14042

Tanda Tangan :

Tanggal : 20 Agustus 2018

Nama : Clara Sonya Angelica Aritonang

NIM : 12S14045

Tanda Tangan :

Tanggal : 20 Agustus 2018

# **HALAMAN PENGESAHAN**

Tugas Akhir ini diajukan oleh :

1. Nama : Yusuf Pangaribuan

Nim : 12S14009

Program Studi : Sarjana Sistem Informasi

1. Nama : Lasria Oselina Nainggolan

Nim : 12S14042

Program Studi : Sarjana Sistem Informasi

1. Nama : Clara Sonya Angelica Aritonang

Nim : 12S14045

Program Studi : Sarjana Sistem Informasi

Judul Tugas Akhir : Pemindaian Memori Proses untuk Pendeteksian

Piranti Lunak Jahat

Telah berhasil dipertahankan dihadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana Teknik pada Program Studi Sarjana Sistem Informasi, Fakultas Teknik Informatika dan Elektro, Institut Teknologi Del.

DEWAN PENGUJI

Pembimbing : Tennov Simanjuntak, S.T., M.Sc. ( )

Penguji : Mario Elyezer Subekti Simaremare, S.Kom., M.Sc. ( )

Penguji : Samuel Indra Gunawan Situmeang, S.TI., M.Sc. ( )

Ditetapkan di : Laguboti

Tanggal : 23 Juli 2018

# **KATA PENGANTAR**

Puji dan syukur kehadirat Tuhan Yang Maha Esa atas berkat dan rahmatNya, sehingga Penulis dapat menyelesaikan Tugas Akhir ini sesuai dengan waktu yang direncanakan. Penulisan Tugas Akhir disusun sebagai bagian dari syarat kelulusan Program Studi Sarjana Sistem Informasi di Institut Teknologi Del. Adapun judul Tugas Akhir ini adalah “Pemindaian Memori Proses untuk Pendeteksian Piranti Lunak Jahat”. Dalam penyusunan dan penulisan Tugas Akhir ini tidak terlepas dari bantuan, bimbingan, serta dukungan dari berbagai pihak. Dalam kesempatan ini, Penulis dengan senang hati menyampaikan terimakasih kepada:

1. Tuhan Yesus Kristus, karena atas berkat dan kuasa-Nya, peneliti dilancarkan untuk menyelesaikan Tugas Akhir dengan baik dan tepat waktu.
2. Kedua orang tua dari ketiga peneliti, atas dukungan spiritual, moril dan materil kepada peneliti dalam pengerjaan Tugas Akhir serta memberikan doa yang tiada berkesudahan kepada peneliti.
3. Bapak Tennov Simanjuntak, S.T., M.Sc., selaku dosen pembimbing yang memberikan banyak saran untuk pendalaman materi dan motivasi serta juga memberikan nasehat, bimbingan dan bantuan dalam pengerjaan Tugas Akhir ini.
4. Bapak Mario Elyezer Subekti Simaremare, S.Kom., M.Sc. dan Bapak Samuel Indra Gunawan Situmeang, S.TI., M.Sc., selaku dosen penguji Tugas Akhir yang memberikan saran dan masukan dalam pengerjaan Tugas Akhir.
5. Bapak Jend. Purn. Luhut Binsar Panjaitan, Pembina Yayasan Del, selaku inspirator peneliti yang sangat peduli terhadap pendidikan di kawasan Danau Toba dan juga selaku pendiri Institut Teknologi Del yang memberikan banyak kontribusi dan perhatian terhadap pendidikan di daerah terpencil.
6. Teman-teman SI01 dan SI02 2014 yang ikut berjuang selama 4 tahun menimba ilmu di Institut Teknologi Del dan juga berjuang dalam penyusunan Tugas Akhir masing-masing serta memberikan canda tawa dan memiliki kesan yang sangat berarti bagi peneliti selama masa perkuliahan.
7. Juga tak lupa kepada semua pihak yang tidak dapat disebutkan satu per satu yang telah membantu peneliti dalam penyelesaian Tugas Akhir ini.

Penulis berharap laporan Tugas Akhir ini dapat bermanfaat bagi semua pihak yang memerlukannya. Penulis juga menyadari bahwa laporan Tugas Akhir ini belum sempurna, sehingga penulis mengharapkan saran dan kritik yang membangun untuk perbaikan laporan Tugas Akhir di masa yang akan datang.

Laguboti, Agustus 2018

Penulis

HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI KARYA ILMIAH UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Institut Teknologi Del, kami yang bertanda tangan di bawah ini:

1. Nama : Yusuf Pangaribuan

NIM : 12S14009

Program Studi : Sistem Informasi

Fakultas : Teknik Informatika dan Elektro

1. Nama : Lasria Oselina Nainggolan

NIM : 12S14042

Program Studi : Sistem Informasi

Fakultas : Teknik Informatika dan Elektro

1. Nama : Clara Sonya Angelica Aritonang

NIM : 12S14045

Program Studi : Sistem Informasi

Fakultas : Teknik Informatika dan Elektro

Jenis Karya : Tugas Akhir

Demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Institut Teknologi Del Hak Bebas Royalti Noneksklusif (*Non-exclusife Royalty-Free Right*) atas karya ilmiah kami yang berjudul:

Pemindaian Memori Proses untuk Pendeteksian Piranti Lunak Jahat

Beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Institut Teknologi Del berhak menyimpan, mengalih/media-formatkan, mengelola dalam bentuk pangkalan data (*database*). Merawat dan mempublikasikan karya akhir kami tanpa meminta izin dari kami selama tetap mencantumkan nama kami sebagai penulis/pencipta dan sebagai pemilik Hak Cipta. Demikian pernyataan ini kami buat dengan sebenarnya.

Dibuat di : Laguboti

Pada tanggal : 20 Agustus 2018

Yang menyatakan,

(Yusuf Pangaribuan) (Lasria O. Nainggolan) (Clara S. A. Aritonang)

**ABSTRAK**

Ada 2 pendekatan dalam pendeteksian *malware*, yaitu analisis statik dan analisis dinamik. Analisis statik mendeteksi *malware* tanpa menjalankan program, sehingga *malware* tidak terdeteksi dengan pendekatan ini. Hal ini dikarenakan saat ini *malware* telah menggunakan teknik obfuskasi. Dari kelemahan analisis statik, peneliti mengekstensi *debugger* sederhana yang menerapkan analisis dinamik. Analisis dinamik mendeteksi *malware* dengan menjalankan program pada lingkungan terkendali sehingga diharapkan semua kode yang diobfuskasi telah terbuka. Pada penelitian ini, pertama sekali mengakses *file* yang diteliti dengan meng-*input* *path file* tersebut, kemudian mencari alamat dari *text se*ction dengan menggunakan *library* PE *file*. Setelah mendapatkan alamat dari *text section*, ada 2 cara untuk menghentikan *debugger* yaitu dengan teknik *timer* dan men-*set* *breakpoint* secara *random*. Setelah *debugger* diberhentikan sementara, maka isi dari memori telah diperoleh dan digunakan pada proses *string* *matching*. Pada proses *string* *matching*, isi memori yang telah didapat sebelumnya akan dibandingkan dengan *database* *signature malware*. Jika isi memori tersebut ada pada *database* maka dapat dipastikan *file executable* tersebut mengandung *malware.* Namun apabila tidak ada, maka belum dapat dipastikan apakah *file* mengandung *malware* atau tidak. Pada percobaan penelitian ini, peneliti menggunakan AVG versi 9.0 menjadi perbandingan dengan *custom debugger* yang di-*extend* oleh peneliti. AVG versi ini digunakan karena versi ini masih menggunakan pendekatan dengan analisis statik dan *custom debugger* yang diekstensi oleh peneliti menggunakan pendekatan analisis dinamik. Dari hasil percobaan yang dilakukan peneliti, perolehan akurasi menggunakan *custom debugger* yang diekstensi peneliti yaitu 66,67 % sedangkan perolehan akurasi menggunakan AVG verisi 9.0 yaitu 13,33%.

**Kata kunci***:* Piranti lunak jahat, pemindaian memori*,* analisis dinamik

***ABSTRACT***

*There are 2 malware detection approaches, namely static analysis and dynamic analysis. Static analysis detects malware without running program, so malware isn’t detected with this approach because currently malware has used obfuscation techniques. This static analysis’ flaw drives researchers to extend simple debugger that implements dynamic analysis. Dynamic analysis detects malware by running the program in controlled environment so it’s expected that all of ofuscated code has been opened. First, accessing file by inputting the file path, then get address of text section using PE file library. After getting the address of text section, there are 2 ways to suspend the debugger by using the timer technique or setting the breakpoint randomly technique. After that, the contents of memory have been obtained will be used in string-matching process. In this process, the contents of memory will be compared with malware signature database. If the contents of memory in database can be ascertained, executable file contains malware. But if it's not there, it can't be ascertained whether the file contains malware or not. On experiment for detecting malware, researchers using AVG version 9.0 as a comparison with a custom debugger that was extended by the researcher. Researchers use this version of AVG because of its static analysis approach and custom debugger which is extended by researchers using a dynamic analysis approach. Experiment’s result conducted by researchers show the accuracy obtained using custom debugger was 66.67% while the accuracy obtained using AVG version 9.0 was 13.33%.*

***Keywords****: Malware, memory scanning, dynamic analysis*

# **DAFTAR ISI**

[HALAMAN JUDUL i](#_Toc522192718)

[HALAMAN PERNYATAAN ORISINALITAS ii](#_Toc522192719)

[HALAMAN PENGESAHAN iii](#_Toc522192720)

[KATA PENGANTAR iv](#_Toc522192721)

[HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI KARYA ILMIAH UNTUK KEPENTINGAN AKADEMIS vi](#_Toc522192722)

[ABSTRAK vii](#_Toc522192723)

[DAFTAR ISI ix](#_Toc522192724)

[DAFTAR TABEL xii](#_Toc522192725)

[DAFTAR GAMBAR xiii](#_Toc522192726)

[DAFTAR PERSAMAAN xiv](#_Toc522192727)

[DAFTAR LAMPIRAN xvi](#_Toc522192728)

[DAFTAR PSEUDOCODE xvii](#_Toc522192729)

[BAB 1 PENDAHULUAN 1](#_Toc522192730)

[1.1 Latar Belakang 1](#_Toc522192731)

[1.2 Rumusan Masalah 3](#_Toc522192732)

[1.3 Tujuan Penelitian 4](#_Toc522192733)

[1.4 Ruang Lingkup dan Batasan Masalah 4](#_Toc522192734)

[BAB 2 LANDASAN TEORI 5](#_Toc522192735)

[2.1 *Malware* 5](#_Toc522192736)

[2.2 Analisis *Malware* 7](#_Toc522192737)

[2.2.1 Analisis Statis 7](#_Toc522192738)

[2.2.2 Analisis Dinamik 8](#_Toc522192739)

[2.3 Obfuskasi 8](#_Toc522192740)

[2.4 *Sandbox* 10](#_Toc522192741)

[2.5 Windows *Portable Executable* (PE) 10](#_Toc522192742)

[2.6 Memori 12](#_Toc522192743)

[2.6.1 Memori Fisik 12](#_Toc522192744)

[2.6.2 Memori Virtual 13](#_Toc522192745)

[2.6.3 *Windows Process Memory* 13](#_Toc522192746)

[2.6.4 Windows *Process Memory* API 14](#_Toc522192747)

[2.6.4.1 Fungsi ReadProcessMemory 14](#_Toc522192748)

[2.6.4.2 Fungsi WriteProcessMemory 15](#_Toc522192749)

[2.6.4.3 Fungsi VirtualProtectEx 16](#_Toc522192750)

[2.6.4.4 Fungsi VirtualQueryEx 16](#_Toc522192751)

[2.7 Windows *Debugger* 17](#_Toc522192752)

[2.7.1 Windows *Debuging* API 18](#_Toc522192753)

[2.7.1.1 Fungsi ContinueDebugEvent 18](#_Toc522192754)

[2.7.1.2 Fungsi DebugActiveProcess 19](#_Toc522192755)

[2.7.1.3 Fungsi DebugActiveProcessStop 19](#_Toc522192756)

[2.7.1.4 Fungsi WaitForDebugEvent 19](#_Toc522192757)

[2.7.2 *Debug Event* 20](#_Toc522192758)

[2.8 *Breakpoint* 22](#_Toc522192759)

[2.8.1 *Software Breakpoint* 22](#_Toc522192760)

[2.8.2 *Hardware Breakpoint* 23](#_Toc522192761)

[2.8.3 *Memory Breakpoint* 24](#_Toc522192762)

[2.9 Konsep *Memory Scanning* 25](#_Toc522192763)

[2.10 *Malware Signature* 25](#_Toc522192764)

[2.10.1 Struktur *Signature* 26](#_Toc522192765)

[2.10.2 CVD (ClamAV *Virus Database*) 27](#_Toc522192766)

[2.11 *Multi-Pattern String Matching* 28](#_Toc522192767)

[2.11.1 Algoritma Boyer-Moore 29](#_Toc522192768)

[2.11.2 Algoritma Rabin-Karp 30](#_Toc522192769)

[2.11.3 Struktur Data Suffix Tree 31](#_Toc522192770)

[2.12 Antivirus 34](#_Toc522192771)

[BAB 3 ANALISIS 35](#_Toc522192772)

[3.1 Analisis Data *Signature Malware* 35](#_Toc522192773)

[3.2 Windows *Custom* *Debugger* 36](#_Toc522192774)

[3.3 Penentuan Titik Penghentian Program atau *Breakpoint* 39](#_Toc522192775)

[3.4 Penentuan Teknik untuk Menghentikan *Debugger* Sementara 41](#_Toc522192776)

[3.5 *Debug Event* 41](#_Toc522192777)

[3.6 Analisis Memori 42](#_Toc522192778)

[3.7 Algoritma *String Matching* 43](#_Toc522192779)

[3.7.1 Algoritma Boyer-Moore 43](#_Toc522192780)

[3.7.1.1 Aturan *Bad Character* 44](#_Toc522192781)

[3.7.1.2 Aturan *Good Shift* 45](#_Toc522192782)

[3.7.2 Algoritma Rabin-Karp 47](#_Toc522192783)

[3.7.3 Struktur Data *Suffix Tree* 48](#_Toc522192784)

[3.7.3.1 Proses Pemenggalan *String* 48](#_Toc522192785)

[3.7.3.2 Proses Pembangunan *Suffix* *Tree* 49](#_Toc522192786)

[3.7.3.3 Proses *String Matching* 50](#_Toc522192787)

[BAB 4 RANCANGAN 51](#_Toc522192788)

[4.1 DFD Rancangan Sistem 52](#_Toc522192789)

[4.1.1 DFD Level 0 52](#_Toc522192790)

[4.1.2 DFD Level 1 52](#_Toc522192791)

[4.1.2.1 Mengakses *File Executable* 54](#_Toc522192792)

[4.1.2.2 *Get Address Text Section* 55](#_Toc522192793)

[4.1.2.3 Mendapatkan *Snapshot Text Section* 55](#_Toc522192794)

[4.1.2.4.5 *Execute Code* 57](#_Toc522192795)

[4.1.2.4 Read Process Memory 59](#_Toc522192796)

[4.1.2.5 Deteksi *Signature* dengan Algoritma *String Matching* 59](#_Toc522192797)

[4.1.2.5.1 Algoritma Boyer-Moore 59](#_Toc522192798)

[4.1.2.5.2 Algoritma Rabin-Karp 60](#_Toc522192799)

[4.1.2.5.3  *Suffix Tree* 61](#_Toc522192800)

[BAB 5 IMPLEMENTASI 62](#_Toc522192801)

[5.1 Batasan Implementasi 62](#_Toc522192802)

[5.2 Implementasi *Debugger* dengan Algoritma *Memory Scanning* 62](#_Toc522192803)

[5.2.1 Implementasi untuk Mengakses *File Executable* 64](#_Toc522192804)

[5.2.2 Implementasi untuk Mengakses *Text Section* 64](#_Toc522192805)

[5.2.3 Implementasi untuk Mengambil *Snapshot* *Text Section* 65](#_Toc522192806)

[5.2.3.1 Memanggil Fungsi Run 67](#_Toc522192807)

[5.2.4 Mengambil *Signature Malware* dari *Database* 69](#_Toc522192808)

[5.2.5 Proses *String Matching* 69](#_Toc522192809)

[5.2.5.1 Algoritma Boyer Moore 69](#_Toc522192810)

[5.2.5.2 Algoritma Rabin-Karp 70](#_Toc522192811)

[5.2.5.3 Suffix Tree 72](#_Toc522192812)

[BAB 6 EKSPERIMEN DAN PEMBAHASAN 73](#_Toc522192813)

[6.1 Pengujian 73](#_Toc522192814)

[6.1.1 Tujuan Pengujian 73](#_Toc522192815)

[6.1.2 Persiapan Pengujian 74](#_Toc522192816)

[6.1.2.1 File Executable 74](#_Toc522192817)

[6.1.2.2 Lingkungan Pengujian 74](#_Toc522192818)

[6.1.2.3 Skenario Pengujian 74](#_Toc522192819)

[6.2 Pengujian *Custom Debugger* 75](#_Toc522192820)

[6.3 Percobaan 76](#_Toc522192821)

[6.4 Pembahasan 81](#_Toc522192822)

[6.4.1 *Custom Debugger* 81](#_Toc522192823)

[6.4.1.1 Boyer-Moore 82](#_Toc522192824)

[6.4.1.2 *Suffix Tree* 82](#_Toc522192825)

[6.4.1.3 Rabin-Karp 83](#_Toc522192826)

[6.4.2 Antivirus AVG 83](#_Toc522192827)

[6.4.3 Hasil Percobaan 84](#_Toc522192828)

[BAB 7 KESIMPULAN DAN SARAN 87](#_Toc522192829)

[7.1 Kesimpulan 87](#_Toc522192830)

[7.2 Saran 88](#_Toc522192831)

[DAFTAR PUSTAKA 89](#_Toc522192832)

# **DAFTAR TABEL**

[Tabel 2.1 Parameter Fungsi ReadProcessMemory 15](#_Toc522045342)

[Tabel 2.2 Parameter Fungsi WriteProcessMemory 15](#_Toc522045343)

[Tabel 2.3 Parameter Fungsi VirtualProtextEx 16](#_Toc522045344)

[Tabel 2.4 Parameter Fungsi VirtualQueryEx 17](#_Toc522045345)

[Tabel 2.5 Parameter Fungsi ContinueDebugEvent 19](#_Toc522045346)

[Tabel 2.6 Parameter FungsiDebugActiveProcess 19](#_Toc522045347)

[Tabel 2.7 Parameter Fungsi DebugActiveProcessStop 19](#_Toc522045348)

[Tabel 2.8 Parameter Fungsi WaitForDebugEvent 20](#_Toc522045349)

[Tabel 2.9. Parameter Struktur DEBUG\_EVENT 20](#_Toc522045350)

[Tabel 2.10 Tipe *Signature* [27] 26](file:///G:\FINALKALIKALIKALI.docx#_Toc522045351)

[Tabel 3.1 Hasil Identifikasi *Malware* 36](#_Toc522045352)

[Tabel 3.2 Pemenggalan *String* 49](#_Toc522045353)

[Tabel 6.1 *Use* *Case* *Scenario* Pengujian *File* yang Diduga *Malware* 75](#_Toc522045354)

[Tabel 6.2 *Use* *Case* *Scenario* Pengujian *File* Tidak Dapat Disimpulkan Sebagai *Malware* 76](#_Toc522045355)

[Tabel 6.3 Percobaan *File* *Executable* 77](#_Toc522045356)

**DAFTAR GAMBAR**

[Gambar 2.1 Format Berkas PE 11](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079691)

[Gambar 2.2 Pemetaan Memori Virtual dan Fisik [10] 12](#_Toc522079692)

[Gambar 2.3 Konsep *Software* *Breakpoint* 23](#_Toc522079693)

[Gambar 2.4 *Tree* *Set* *String* [39] 32](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079694)

[Gambar 2.5 *Compressed* *Tree* [39] 33](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079695)

[Gambar 2.6 *Suffix Tree* [37] 33](#_Toc522079696)

[Gambar 3.1 Sigtool *Command* 36](#_Toc522079697)

[Gambar 3.2 Kode Program *Load* *Custom* *Debugger* 38](#_Toc522079698)

[Gambar 3.3 Kode Program Inisialisasi *Variable* 39](#_Toc522079699)

[Gambar 3.4 Kode Program Open\_Process *Custom Debugger* 39](#_Toc522079700)

[Gambar 3.5 Kode Program *run() Custom Debugger* 39](#_Toc522079701)

[Gambar 3.6 Contoh Algoritma Boyer-Moore 43](#_Toc522079702)

[Gambar 3.7 Contoh *Bad* *Character* 44](#_Toc522079703)

[Gambar 3.8 Kasus 1 *Good* *Shift* 45](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079704)

[Gambar 3.9 Kasus 2 *Good* *Shift* 46](#_Toc522079705)

[Gambar 3.10 Kasus 3 *Good* *Shift* 47](#_Toc522079706)

[Gambar 3.11 Suffix *Tree* 50](#_Toc522079707)

[Gambar 4.1 Arsitektur Sistem 51](#_Toc522079708)

[Gambar 4.2 DFD Level 0 52](#_Toc522079709)

[Gambar 4.3 DFD Level 1 53](#_Toc522079710)

[Gambar 4.4 *Flowchart* Mengakses *File* *Executable* 54](#_Toc522079711)

[Gambar 4.5 *Flowchart Timer* 56](#_Toc522079712)

[Gambar 4.6 *Flowchart* Penentuan *Breakpoint* secara Acak 57](#_Toc522079713)

[Gambar 4.7 *Flowchart Execute Code* 58](#_Toc522079714)

[Gambar 4.8 *Flowchart* *Suspend* *Debugger* 58](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079715)

[Gambar 4.9 *Flowchart* Suffix *Tree* 61](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079716)

[Gambar 5.1 Implementasi main() 63](#_Toc522079717)

[Gambar 5.2 Implementasi *Load Process* 64](#_Toc522079718)

[Gambar 5.3 Implementasi getAddress() 65](#_Toc522079719)

[Gambar 5.4 Implementasi Teknik *Timer* 65](#_Toc522079720)

[Gambar 5.5 Implementasi Teknik *Set* *Breakpoint* dengan *Random* 66](#_Toc522079721)

[Gambar 5.6 implementasi bp\_set 66](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079722)

[Gambar 5.7 Implementasi *Read* *Process* *Memory* 66](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079723)

[Gambar 5.8 Implementasi write\_process\_memory 67](#_Toc522079724)

[Gambar 5.9 Implementasi exception\_handler\_breakpoint() 67](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079725)

[Gambar 5.10 Implementasi fungsi run() 68](#_Toc522079726)

[Gambar 5.11 Implementasi get\_debug\_event() 68](#_Toc522079727)

[Gambar 5.12 Implementasi suspend() 68](#_Toc522079728)

[Gambar 5.13 Implementasi getFromDatabase() 69](#_Toc522079729)

[Gambar 5.14 Implementasi Boyer-Moore 69](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079730)

[Gambar 5.15 Implementasi Boyer-Moore dengan pendekatan *Bad* *Character* *Heuristic* 70](#_Toc522079731)

[Gambar 5.16 Implementasi Algoritma Rabin Karp 71](file:///C:\Users\Ramosan\Desktop\FINALKALIKALIKALI.docx#_Toc522079732)

[Gambar 5.17 Implementasi Pendefenisian Fungsi Rabin Karp 71](#_Toc522079733)

[Gambar 5.18 Implementasi detach() 71](#_Toc522079734)

[Gambar 5.19 Implementasi Algoritma *Suffix Tree* 72](#_Toc522079735)

[Gambar 6.1 *Screenshot* Pendeteksian AVG 80](#_Toc522079736)

[Gambar 6.2 *Screenshot* Pendeteksian fax.exe 84](#_Toc522079737)

[Gambar 6.3 *Screenshot* Pendeteksian trojaned.exe 85](#_Toc522079738)

[Gambar 6.4 *Screenshot* Pendeteksian win33.exe 85](#_Toc522079739)

[Gambar 6.5 *Screenshot* Pendeteksian trojan\_encoded.exe 86](#_Toc522079740)

[Gambar 6.6 Hasil Memori Boyer-Moore 86](#_Toc522079741)

[Gambar 6.7 Hasil Memori Rabin-Karp 86](#_Toc522079742)

[Gambar 6.8 Hasil Memori Suffix Tree 86](#_Toc522079743)

# **DAFTAR PERSAMAAN**

[( 2.1 ) 30](#_Toc522024581)

[( 6.1 ) 79](#_Toc522024582)

[( 6.2 ) 80](#_Toc522024583)

# **DAFTAR LAMPIRAN**

[Lampiran 1: *File Executable* berisi *Malware* i](#_Toc522132018)

**DAFTAR PSEUDOCODE**

[*Pseudocode* 4.1 *Create Process* 54](file:///F:\FINALKALIKALIKALI.docx#_Toc521959365)

[*Pseudocode* 4.2 *Get Address Text Section* 55](file:///F:\FINALKALIKALIKALI.docx#_Toc521959366)

[*Pseudocode* 4.3 Menentukan *Breakpoint* 56](file:///F:\FINALKALIKALIKALI.docx#_Toc521959367)

[*Pseudocode* 4.4 *Suspend* *Process* 59](file:///F:\FINALKALIKALIKALI.docx#_Toc521959368)

[*Pseudocode* 4.5 *Read* *Process* *Memory* 59](file:///F:\FINALKALIKALIKALI.docx#_Toc521959369)

[*Pseudocode* 4.6 Algoritma Boyer-Moore 60](file:///F:\FINALKALIKALIKALI.docx#_Toc521959370)

[*Pseudocode* 4.7 Algoritma Rabin Karp 61](file:///F:\FINALKALIKALIKALI.docx#_Toc521959371)

# **PENDAHULUAN**

Pada bab ini diuraikan latar belakang, tujuan penelitian, ruang lingkup dan arsitektur sistem yang dibangun.

* 1. Latar Belakang

Perkembangan piranti lunak jahat atau *malware* semakin kompleks dimana *attacker* membuat kode-kode baru yang sulit dideteksi oleh pendeteksi *malware* [1]. Ada 2 pendekatan dalam pendeteksian *malware*, yakni analisis statis dan analisis dinamik. Pada analisis statik, pendeteksian *malware* dilakukan tanpa menjalankan *malware* tersebut. Dengan pendekatan ini, pendeteksian lebih cepat dan tidak berbahaya. Tetapi pada saat ini, analisis statik bisa dikalahkan dengan menerapkan teknik obfuskasi atau teknik – teknik untuk penghindaran deteksi. Pendekatan dengan analisis dinamik, dimana program tersebut dijalankan di lingkungan yang terkendali. Pada pendekatan ini penentuan *malware* atau tidak, ditentukan berdasarkan jejak eksekusi.

Para *attacker* terus menerapkan teknik-teknik penghindaran pendeteksian *malware* pada *executable*. Teknik umum yang dilakukan para pembuat *malware* adalah kompresi (*packing*) atau enkripsi (*encryption*). *Malware* juga melakukan teknik obfuskasi untuk penghindaran pendeteksian sehingga pendekatan dengan analisis statik tidak dapat mendeteksinya. Untuk mengatasi kekurangan analisis statik tersebut maka teknik yang lazim digunakan yaitu analisis dinamik. Pada analisis dinamik *file* *executable* dijalankan pada lingkungan terkendali dan perilaku-perilaku *malware* tersebut bisa dikenali melalui isi memori [2].

Salah satu tantangan untuk melakukan pemindaian memori proses adalah merancang algoritma yang dapat digunakan untuk menentukan kapan pengambilan isi memori proses atau disebut sebagai pengambilan *snapshot* dilakukan. Tantangan lain adalah bagaimana pemindaian memori atau *memory scanning* dari *executable* yang sedang dieksekusi dibaca dan dicocokkan dengan sekumpulan tanda atau *signature malware* yang ada pada basis data secara efisien.

Pada penelitian ini pengembangan *debugger* tidak dilakukan dari awal, tetapi pengembangan dilakukan dengan mengidentifikasi dan mempelajari *debugger* yang telah ada yang sudah mampu melakukan instrumentasi biner dari sebuah *executable* yang sedang dianalisis. Instrumentasi dalam istilah pemograman adalah tindakan menanamkan kode *monitoring* ke dalam sistem perangkat lunak. Salah satu teknik nstrumentasi dapat dilakukan dengan cara transformasi biner program. Instrumentasi biner pada penelitian ini dilakukan dengan memanipulasi saat program di tahap sudah dimuat dalam memori. Dalam pengertiannya, instrumentasi merupakan penambahan kode kepada suatu program yang dianalisis. Instrumentasi dikelompokkan kedalam dua jenis yaitu instrumentasi biner statik dan instrumentasi biner dinamik. Instrumentasi biner statik dilakukan pada saat sebelum program dijalankan dengan cara menulis kembali kode program dari *executable*. Sementara, instrumentasi biner dinamik dilakukan pada saat *runtime* dengan cara menganalisa perilaku *executable* ketika instrumentasi. *Debugger* merupakan *tool* yang melakukan *memory scanning* pada saat *runtime* sehingga penelitian ini akan menerapkan instrumentasi biner dinamik. Oleh sebab itu, peneliti juga perlu memahami bagaimana cara kerja sebuah *custom debugger* dan cara untuk mengembangkannya atau mengekstensinya sehingga algoritma untuk pengambilan isi memori dan pemindaian tanda dapat dilakukan dengan efisien.

Dengan dieksekusinya *malware* tersebut pada sebuah *custom debugger*, maka diharapkan semua kode-kode atau prosedur yang digunakan untuk deobfuskasi akan dieksekusi oleh sebuah *executable* sehingga isi memori khususnya pada *text section* dapat dipindai untuk mencari tanda-tanda *malware*. *Memory scanning* sendiri dapat dianggap masalah pencarian *string* atau *string searching* dimana pada penelitian ini string berupa heksadesimal dari instruksi komputer (*opcode*). Oleh sebab itu, salah satu tantangan lain dalam penelitian ini adalah mendapatkan algoritma pemindaian memori yang efisien.

Pada Tugas Akhir ini, peneliti berkontribusi terhadap peningkatan sistem analisis dinamik untuk pendeteksian *malware*, khususnya pada penerapan *in memory process scanning*. Algoritma *memory scanning* ini penting dilakukan karena secanggih apapun metode deteksi semantik yang dilakukan, pembuat *malware* tersebut pasti selalu mempunyai cara untuk mengatasi pendeteksian dengan melakukan transformasi atau obfuskasi. Proses *memory scanning* dilakukan dengan menjalankan m*alware* namun menjalankan *malware* secara langsung merupakan tindakan berbahaya karena *malware* bisa menginfeksi mesin di perangkat lainnya dimana perangkat yang berinteraksi langsung dengan perangkat yang terinfeksi *malware*. Untuk mengatasi hal tersebut, *executable* yang terinfeksi *malware* akan dianalisis pada *virtual box* sehingga *malware* hanya dapat merusak virtual PC bukan aslinya, satu contoh *virtual box* yang popular adalah VMWare [2]. Oleh sebab itu, *custom debugger* akan dijalankan berdasarkan perspektif peneliti atau analis untuk menganalisis *malware* yang dijalankan diatas *virtual* *box* atau VMWare. Beberapa *debugger* yang sudah dikembangkan berbasis Windows, baik berbayar contohnya IDA Pro maupun *debugger* yang tidak berbayar contohnya Immunity Debugger dan Olly Debugger. Pada penelitian ini, peneliti memutuskan untuk mengembangkan *debugger* sesuai dengan kebutuhan yang mampu melakukan pemindaian memori. *Debugger* yang dibangun tidak berdasarkan *debugger-debugger* sebelumnya karena *debugger* tersebut memiliki fitur-fitur tertentu yang tidak dibutuhkan peneliti pada *debugger* yang dikembangkan. Selain itu, dengan mengembangkan *custom debugger,* kelompok peneliti memiliki total kontrol terhadap struktur dari aplikasi yang dibangun sehingga memudahkan peneliti untuk membangun sistem *memory scanning*.’

* 1. Rumusan Masalah

Rumusan masalah pada penelitian Tugas Akhir ini adalah:

1. Bagaimana rancangan algoritma yang dapat digunakan untuk mengambil isi memori proses yang sedang dianalisis sehingga diharapkan semua kode deobfuskasinya telah selesai dieksekusi?
2. Bagaimana mengekstensi atau mengembangkan *custom debugger* yang mampu melakukan instrumentasi biner sehingga pengembangan pemindaian memori proses dapat dilakukan?
3. Algoritma mana yang paling baik pada pemindaian memori dalam mendeteksi *executable* yang jahat berdasarkan sekumpulan tanda yang ada di basis data?
   1. Tujuan Penelitian

Tujuan dari pelaksanaan Tugas Akhir ini adalah:

1. Mengembangkan sistem yang berguna untuk melakukan pendeteksian *malware* secara dinamik dengan metode *in memory process scanning* yang mampu untuk menentukan kapan pengambilan memori dilakukan sehingga diharapkan kode deobfuskasinya telah selesai dieksekusi*.*
2. Merancang dan menerapkan algoritma penghentian proses untuk pengambilan memori
3. Menganalisis algoritma pemindaian memori yang bekerja berdasarkan masalah pencarian *string* untuk mendeteksi jahat atau tidaknya *executable* yang sedang dianalisis.
   1. Ruang Lingkup dan Batasan Masalah

Ruang lingkup pada pengerjaan Tugas Akhir ini adalah:

1. Objek penelitian yang digunakan adalah *file executable* dalam sistem operasi Windows*.*
2. Tugas akhir ini mengekstensi *debugger* yang dibangun oleh Justin Seitz [3].
3. Bagian memori yang diambil dan dipindai adalah *text section* adapun alasannya adalah tanda-tanda *malware* ditemukan pada *string* berupa heksadesimal dari instruksi komputer.
4. Penelitian ini belum menangani teknik-teknik penghindaran deteksi, seperti obfuskasi atau penanda jika dijalankan di *debugger*.

# **LANDASAN TEORI**

Beberapa teori yang dijadikan landasan pengerjaan Tugas Akhir akan dibahas pada bab ini. Definisi dan konsep yang diperoleh dari sumber literatur akan digunakan untuk memberikan penjelasan dengan rinci dan faktual.

* 1. *Malware*

*Malware* (*Malicious Software*) merupakan piranti lunak jahat yang menyerang piranti lunak lainnya, dengan tujuan merusak dan mengubah perilaku yang sebenarnya menjadi perilaku yang diinginkan oleh *malware* itu [4].

*Malware* diidentifikasi kedalam 3 karakteristik, yaitu:

1. Mampu melakukan replikasi diri. *Malware* berusaha melakukan penyebaran diri dengan membuat salinan baru, atau menginstansiasi dirinya sendiri. *Malware* juga dapat memperbanyak diri melalui *user* yang menyalinnya secara tidak sengaja.
2. Mampu melakukan pertumbuhan atau penambahan populasi. Pertumbuhan atau penambahan populasi *malware* terjadi jika terdapat perubahan dari instansi *malware* akibat replikasi diri. *Malware* yang tidak melakukan replikasi diri memiliki pertumbuhan populasi nol (tidak ada).
3. Berperilaku sebagai parasit. *Malware* menggunakan kode *executable* lain sebagai media untuk menyerang sistem agar keberadaannya diketahui. Arti “*executable*” ialah objek yang dapat dieksekusi, seperti kode blok *boot* *disk*, kode biner dalam aplikasi, dan kode yang diinterpretasi. Termasuk juga *source code*, seperti aplikasi *scripting languages*, dan kode yang memerlukan kompilasi sebelum dieksekusi.

Secara garis besar, *malware* memiliki 4 tahap siklus hidup, yaitu [5]:

1. *Dormant phase*

Pada fase ini *malware* tidaklah aktif. *Malware* akan diaktifkan oleh suatu kondisi tertentu, misalnya tanggal yang ditentukan, kehadiran program lain/dieksekusinya program lain dan sebagainya. Tidak semua *malware* melalui fase ini.

1. *Propagation phase*

Pada fase ini, *malware* akan mereplikasikan dirinya kepada suatu program atau ke suatu tempat dari media *storage* (baik *hardisk*, RAM). Setiap program yang terinfeksi akan menjadi hasil kloning dari *malware* (tergantung cara dari setiap *malware* dalam menginfeksi program).

1. *Trigerring phase*

Pada fase ini, akan ada beberapa kejadian sistem yang memicu *malware* untuk aktif melakukan fungsi tertentu. Seperti pada fase dormant, ada beberapa kondisi yang memicu sehingga pada fase ini *malware* yang tidak aktif menjadi aktif.

1. *Execution phase*

Di fase ini, *malware* yang telah aktif akan melakukan fungsinya. Seperti menghapus *file*, menampilkan pesan-pesan, dan sebagainya.

*Malware* dapat dikategorikan beberapa jenis berdasarkan metode operasinya, yaitu [6]:

1. *Logic Bomb,* merupakan *malware* berupa kode yang mengandung *payload* yang memiliki efek jahat serta *trigger payload* yang merupakan kondisi *boolean* mengatur kapan *payload* dieksekusi.
2. *Trojan Horse,* merupakan *malware* berupa program yang sepertinya melakukan beberapa pekerjaan baik (menjalankan sesuai program yang ada) tetapi sebenarnya melakukan pekerjaan-pekerjaan jahat secara tersembunyi. Contohnya adalah mendapatkan *password* *login* dengan menunggu *user* mengetik informasi “*username*” dan “*password*”.
3. *Back Door,* merupakan suatu mekanisme untuk menghindari pemeriksaan sistem keamanan. Salah satu contoh adalah RAT[[1]](#footnote-1), yang mampu memonitor dan mengkontrol komputer dari jarak jauh.
4. *Virus,* merupakan sebuah program yang melakukan replikasi dirinya sendiri kedalam *executable* lainnya dan memiliki potensi yang besar untuk melakukan manipulasi terhadap sistem tersebut.
5. *Worm*, memiliki sifat menyerupai *virus* namun replikasi diri *worm* dibagi atas dua cara, pertama, *worm* sebagai *standalone* yang tidak bergantung pada kode *executable* lain, kedua, *worm* yang mampu menyebar dari mesin ke mesin melalui jaringan.
6. *Rabbit*, merupakan *malware* yang mampu menggandakankan diri dengan cepat.
7. *Spyware,* merupakan suatu piranti lunak yang mampu mengumpulkan informasi dari komputer dan mentransmisikannya kepada pihak lain. Informasi yang dikumpulkan oleh *spyware* memiliki nilai seperti *username* dan *password*, alamat *email*, akun *bank*, serta kode lisensi piranti lunak.
8. *Adware,* memiliki sifat yang serupa dengan *spyware* namun *adware* berfokus pada bidang *marketing*, yang memiliki kemampuan untuk memunculkan iklan atau mengarahkan *web* *browser* *user* ke halaman situs *web* tertentu dengan tujuan untuk mendapatkan *customer*.
9. *Hybrids, droppers, and blended threats. Hybrid* merupakan *malware* yang melakukan perilaku yang jahat dengan mengambil informasi tertentu dari *user*. *Dropper* merupakan *malware* yang menjadi jalur bagi *malware* lain untuk melakukan penyerangan. *Blended threat* merupakan virus yang mengeksploitasi teknik yang rentan dari antivirus.
   1. Analisis *Malware*

Terdapat dua pendekatan utama untuk mendeteksi *malware* yaitu analisis statis dan analisis dinamik.

* + 1. Analisis Statis

Analisis statis merupakan teknik menganalisis *malware* dengan tidak benar-benar menjalankan *malware* tersebut, dalam arti memeriksa kode tanpa mengeksekusi aplikasi melainkan dengan melakukan *disassembly* terhadap kode binerdari *executable* *malware* itu. Proses analisis statik melakukan analisis *malware* dengan menggunakan *disassemble tools* yang digunakan untuk mendapatkan *file* program berbahasa *assembly.* Analisis statik dibagi ke dalam dua bagian, yaitu analisis statik berdasarkan sintaks dan semantik. Analisis statik berdasarkan sintaks mengidentifikasi *malware* berdasarkan sebuah sintaks, yakni sekumpulan kode terurut. Analisis statik berdasarkan semantik mengandalkan pemahaman tentang struktur semantik dari program. Setiap kode biner yang berbeda menunjukkan tipe *malware* yang berbeda juga*.* Keuntungan dari analisis statik adalah komputer lebih aman karena *malware* tidak dieksekusi, sehingga teknik ini menganalisis semua eksekusi biner, dan waktu untuk menganalisis serta mendeteksi *malware* relatif cepat. Namun, analisis statik memiliki kelemahan yaitu sulit mendeteksi program yang dirancang khusus untuk menghindari analisis statik dengan menerapkan sistem teknik obfuskasi dan enkripsi [2].

* + 1. Analisis Dinamik

Analisis dinamik merupakan teknik analisis *malware* yang dilakukan dengan memantau kerentanan dan perilaku program saat program dijalankan. Analisis dinamik menjawab permasalahan dari analisis statis, yang hanya menganalisis instruksi jahat dari *malware*. Analisis dinamik juga mampu menganalisis perilaku *malware* secara akurat dengan memeriksa instruksi jahat secara langsung. Oleh karena itu, analisis dinamik mampu mengatasi percobaan obfuskasi dari *malware* tersebut.

Namun, menjalankan *malware* secara langsung merupakan tindakan berbahaya karena kode jahat bisa menginfeksi mesin di perangkat lainnya. Untuk mengatasi hal tersebut, peneliti akan menjalankan *malware* pada perangkat lunak virtualisasi atau virtualization software yaitu *virtual box* sehingga *malware* hanya dapat merusak virtual PC bukan aslinya [2].

* 1. Obfuskasi

Secara umum, teknik obfuskasi dilakukan untuk membuat program sulit dipahami. Teknik obfuskasi sering digunakan pada *malware* untuk menghindari pendeteksian *malware* dari detektor. Cara kerjanya dengan mengubah/mengacak kode suatu aplikasi namun tetap memelihara semantiknya. Dalam mengacak kode sumber suatu aplikasi, obfuskator dapat menerapkan berbagai algoritma. Teknologi ini biasa digunakan untuk melindungi kekayaan intelektual dari pengembang perangkat lunak dengan melindungi aplikasi program dari *reverse engineering*, namun para pencipta *malware* menyalahgunakan teknologi ini untuk menghindari analisis statik terutama deteksi antivirus [7].

Ada tiga pendekatan yang dilakukan *malware* untuk menghindari pendeteksian antivirus, yaitu: [8]

* + - 1. *Encrypted Malware.* Jenis pendekatan untuk menghindari antivirus dengan melakukan enkripsi *signature malware*. Enkripsi adalah sebuah proses yang mengubah kode yang bisa dimengerti menjadi kode yang tidak dapat dimengerti. Dengan metode enkripsi *signature* *malware*, *detector* tidak dapat mendeteksi *malware* tersebut.
      2. *Oligomorphic dan Polimorphic Malware.* Pada pendekatan ini, *malware* oligomorfikmampu melakukan mutasi *decryptor, decryptor* yang dimaksud merupakan cara sederhana dengan menggunakan dekripsi dan memilih secara acak sehingga membuat pendeteksian dengan pola menjadi lebih sulit. Sementara *malware* polimorfik merupakan virus yang terenkripsi yang dapat mengubah dekripsi setiap generasi. Dimana virus ini menciptakan varian dari dirinya sendiri dengan menggunakan mekanisme enkripsi yang berbeda disetiap generasi yang menghaslkan dekripsi yang berbeda dengan demikian secara efektif dapat melawan *detector* dalam mencari signature dari *descriptor.*
      3. *Metamorphic Malware.* Pendekatan *malware* ini sangat sulit dideteksi oleh antivirus karena *malware* memanfaatkan teknik obfuskasi untuk memanipulasi dirinya untuk terlihat berbeda.Teknik obfuskasi yang digunakan seperti pertukaran *register* (menggunakan register yang berbeda pada setiap generasi), *swap* instruksi (mengganti instruksi dengan instruksi yang setara), permutasi, transposisi (menata ulang instruksi) dan penyisipan *dead-code* dengan menambahkan NOP (*No Operation Performed*).

Ada enam teknik dasar obfuskasi yang digunakan oleh *malware* polimorfik dan metamorfik, yaitu: [8]

1. *Dead-Code Insertion*, merupakan teknik sederhana dengan menyisipkan sebuah instruksi program berupa kode *No Operation Performed* (NOP).
2. *Register Reassignment*, merupakan teknik sederhana dengan menggantikan register yang tidak digunakan dengan register berisi kode *malware*. Pada metode ini kode program dan perilaku program tetap sama.
3. *Subroutine Reordering*, merupakan metode yang dilakukan dengan mengubah urutan subrutin di program secara acak, dan membuat *signature malware* yang berbeda untuk setiap variasi subrutin.
4. *Instruction Substitution*, melakukan substitusi instruksi mengubah kode asli dengan mengganti beberapa instruksi dengan instruksi lain dengan fungsi yang sama.
5. *Code Transposition*, merupakan metode yang dilakukan dengan mengatur urutan instruksi dari kode asli tanpa mempengaruhi perilaku kode program.
6. *Code Integration*, merupakan metode yang dilakukan dengan menyisipkan *brief* baru ke kode sumber program untuk menjalankan kode jahat.
   1. *Sandbox*

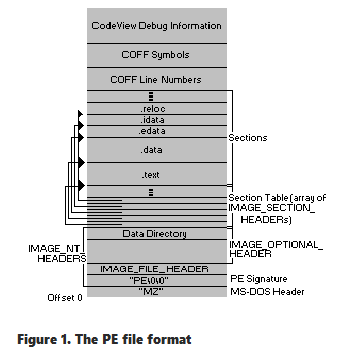
*Sandbox* atau *virtual machine* merupakan teknik virtualisasi pada keamanan sistem dimana pada prinsipnya sebuah *environment* menggunakan proses terbatas yang berjalan menyerupai sistem *environment* aslinya serta dapat dikontrol dengan menggunakan *resource* dari luar sistem. Sehingga ketika *malware* dieksekusi, kerusakannya berdampak pada *environment* virtualnya bukan pada sistem aslinya. Teknik virtualisasi ini digunakan pada keamanan sistem sebagai *environment* untuk menganalisa *routine* dari kode jahat atau *malware*.

Akan tetapi pada saat ini, telah ada teknik penghindaran *sandbox* yang diciptakan pembuat *malware* sehingga *malware* mampu mengatasi teknik *sandbox.* Kaitan *sandbox* dengan analisis dinamik sendiri menjelaskan bahwa telah ada *malware* canggih yang mampu mengatasi analisis dinamik yang menganalisa *malware* menggunakan *virtual* *machine* [9].

* 1. Windows *Portable Executable* (PE)

Windows *Portable Executable* (PE) merupakan sebuah format *file* untuk berkas *executable* yang digunakan oleh sistem operasi Windows versi 32-bit dan 64-bit. PE terbentuk ketika dilakukan kompilasi dan *linking* dari kode sumber program. Terlihat dari Gambar 2.1 pada halaman selanjutnya bahwa struktur data *Portable Executable* (PE) dapat dikelompokkan kedalam dua bagian besar yaitu *header* yang berisi informasi seperti lokasi dan ukuran dari *section-section* yang berisi kode.

*Portable Executable* (PE) ini dikembangkan dari COFF (*Common Object File Format*). COFF adalah bentuk dasar dari *file* obyek (OBJ *file/object file*) milik Linux [10].

**Windows *Portable Executable* (PE) memiliki banyak *section* yang jumlahnya tidak terbatas tergantung dari *compiler* atau *linker* yang dibuat. Berikut ini dijelaskan *section* dari struktur *Portable Executable* (PE).

**Gambar 2.1 Format Berkas PE**

* MS-DOS *header* (IMAGE\_DOS\_HEADER). MS-DOS *header* merupakan suatu struktur dari IMAGE\_DOS\_HEADER.
* *File* *header* (IMAGE\_FILE\_HEADER). *File header* merupakan bentuk struktur dari IMAGE\_FILE\_HEADER yang berada tepat setelah IMAGE\_NT\_SIGNATURE.
* *Optional header* (IMAGE\_OPTIONAL\_HEADER). Struktur ini didefinisikan sebagai IMAGE\_OPTIONAL\_HEADER yang ukurannya 224 BYTE. Struktur *Optional Header* memiliki 31 *field* namun tidak semua dipergunakan.
* *Section Directory*; adalah bagian dari PE yang merupakan isi dari program, serta berada setelah *Optional Header*. Terdiri dari dua bagian utama, yaitu IMAGE\_SECTION\_HEADER dan *Raw Section Data*. IMAGE\_SECTION\_HEADER terdiri dari .reloc , .idata, .edata, .data dan .text.

*Section* .reloc merupakan relokasi tabel untuk instruksi kode yang diinisialisasi yang diperlukan jika *loader* tidak dapat memuat *file*.

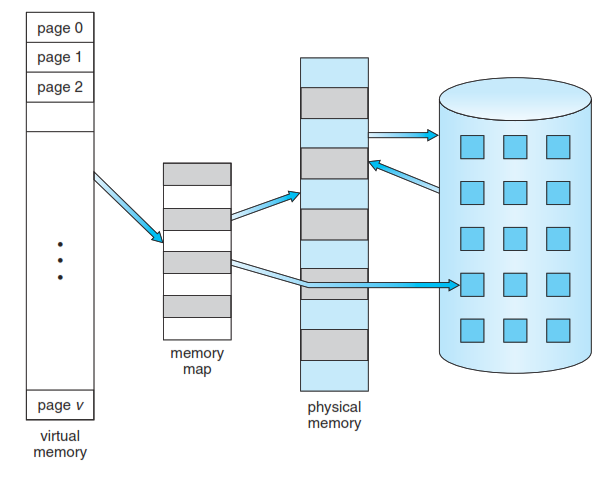
*Section* .idata berisi informasi fungsi dari modul yang di-*import*.

*Section* .edata berisi *list* fungsi dan data dimana modul PE *file* diekspor.

*Section* .data berisi data yang diinisialisasi termasuk global dan statik variabel.

*Section* .text berisi instruksi yang akan dieksekusi.

* 1. Memori

Memori merupakan pusat operasi pada sistem komputer yang menyimpan data serta instruksi yang dibutuhkan CPU (*Central Processing Unit*). Pemetaan dari memori fisik dan memori virtual dilakukan oleh MMU (*Memory-Management Unit*) dapat dilihat pada Gambar 2.2 [10].

**Gambar 2.2 Pemetaan Memori Virtual dan Fisik [10]**

* + 1. Memori Fisik

Pada sistem operasi, memori fisik menyimpan data dan instruksi program Alamat fisik adalah alamat memori yang sebenarnya. Pada saat kompilasi dan waktu pemanggilan, alamat fisik dan alamat logika itu sama. Sedangkan, pada waktu eksekusi menghasilkan alamat fisik dan alamat virtual yang berbeda.

Kumpulan alamat fisik yang berkorespondensi dengan alamat virtual disebut *physical address space.* Alamat virtual harus dipetakan ke alamat fisik sebelum digunakan. Manajemen memori fisik dan memori virtual menggunakan teknik *swapping* yang bekerja ketika program sedang dieksekusi. Teknik *swapping* merupakan pertukaran suatu proses sementara keluar memori ke sebuah penyimpanan sementara dan kemudian dibawa kembali ke memori untuk melanjutkan pengeksekusian. Sistem MMU pada hal ini akan mengkonversikan alamat virtual kedalam alamat fisik dari program yang ada [10].

* + 1. Memori Virtual

Memori virtual merupakan suatu teknik memisahkan memori logis dari memori fisiknya. Memori virtual dapat menampung program dalam skala besar, melebihi daya tampung dari memori utama yang tersedia. Setiap program yang dieksekusi akan membutuhkan akses ke dalam memori. Alamat yang dihasilkan oleh CPU disebut alamat logika. Alamat logika dapat juga disebut sebagai alamat virtual. Lokasi alamat virtual pada manajemen memori disimpan pada memori dan *disk*. Kumpulan alamat virtual dihasilkan oleh CPU disebut *virtual address space*. *Virtual address space* merupakan gambaran secara virtual tentang bagaimana proses disimpan dalam memori. Alamat *virtual* mengatur *page table* dimana *page table* digunakan untuk memetakan alamat *virtual* ke alamat fisik. Implementasi alamat memori dan alamat fisik pada *virtual* memori terjadi ketika program mengakses memori. Cara kerjanya yaitu *processor* akan mengakses *virtual address space* program kemudian dispesifikasikan alamat *virtual*-nya. Setelah mengetahui alamat *virtual* program, selanjutnya memetakan alamat *virtual* ke alamat fisik pada memori, namun apabila alamat fisik tidak berada di memori, maka *page table* akan memetakan alamat *virtual* ke *disk*. Setelah membaca alamat fisik program, maka data akan dikembalikan ke program. Pada memori *virtual*, memori terlihat lebih besar daripada ukuran memori, pembagian memori pada setiap proses sama dan setiap proses didalam sistem terlindung dari proses-proses lainnya [10].

* + 1. *Windows Process Memory*

Proses adalah program yang sedang dieksekusi di memori. Program aplikasi sendiri dapat menggunakan satu atau lebih proses. Setiap proses menyediakan *resource* yang dibutuhkan untuk menjalankan suatu program. *Handle* proses merupakan nilai *integer* yang mengidentifikasikan proses ke Windows. Win32 API menyebutnya sebuah *HANDLE*. Sebagai contoh *handle* ke windows disebut HWND.

Sebuah proses memiliki *virtual address space*, *executable code*, *open handles* ke obyek sistem dimana *term* ‘*handle’* artinya *pointer*, *security context* dimana kumpulan kredensial yang diberikan kepada proses atau akun pengguna yang menciptakan proses, *unique process identifier*, *environment variables*, *priority class* pada penjadwalan prioritas, *minimum* dan *maximum working set sizes*, dan paling tidak memakai satu *thread* pada *execution* [12]*.*

Ketika proses dieksekusi, keadaan proses juga akan berubah. Keadaan proses tersebut didefinisikan berdasarkan aktivitas yang sedang berlangsung terhadap proses tersebut. Berikut jenis-jenis keadaan proses:

* *New* adalah keadaan proses ketika dibentuk pertama sekali.
* *Running* adalah keadaan ketika instruksi dieksekusi.
* *Waiting* adalah keadaan ketika proses sedang menunggu suatu *event* muncul seperti sinyal I/O.
* *Ready* adalah keadaan ketika proses menunggu ditugaskan ke suatu *processor*.
* *Terminated* adalah keadaan ketika proses telah selesai dieksekusi
  + 1. Windows *Process Memory* API

Terdapat beberapa fungsi memori API yang digunakan dalam *memory scanning*.

#### **Fungsi ReadProcessMemory**

Fungsi ini digunakan pada *debugger* untuk dapat mengakses dan membaca alamat memori *virtual* pada suatu proses. Berikut sintaks C++ fungsi ini [13].

BOOL WINAPI ReadProcessMemory (

\_In\_ HANDLE hProcess,

\_In\_ LPCVOID lpBaseAddress,

\_Out\_ LPVOID lpBuffer,

\_In\_ SIZE\_T nSize,

\_Out\_ SIZE\_T \*lpNumberOfBytesRead);

Parameter dan keterangan parameter yang dibutuhkan untuk fungsi ini dapat dilihat pada Tabel 2.1.

**Tabel 2.1 Parameter Fungsi ReadProcessMemory**

|  |  |
| --- | --- |
| Parameter | Keterangan |
| hProcess | Sebuah *handle* ke proses yang memorinya yang sedang dibaca. |
| lpBaseAddress | *Pointer* ke alamat dasar dalam proses yang ditentukan untuk dibaca |
| lpBuffer | *Pointer* ke *buffer* yang menerima konten dari ruang alamat pada proses tertentu. |
| nSize | Jumlah *byte* yang akan dibaca dari proses tertentu. |
| \*lpNumberOfBytesRead | *Pointer* ke variabel yang menerima jumlah *byte* dikirim ke *buffer* tertentu. |

#### **Fungsi WriteProcessMemory**

Fungsi ini digunakan pada *debugger* untuk dapat mengakses dan menulis di alamat memori *virtual* pada proses tertentu. Berikut sintaks C++ fungsi ini [14].

BOOL WINAPI WriteProcessMemory(

\_In\_  HANDLE  hProcess,

\_In\_  LPVOID  lpBaseAddress,

\_In\_  LPCVOID lpBuffer,

\_In\_  SIZE\_T  nSize,

\_Out\_ SIZE\_T  \*lpNumberOfBytesWritten

);

Parameter dari fungsi ini dapat dilihat pada Tabel 2.2.

**Tabel 2.2 Parameter Fungsi WriteProcessMemory**

|  |  |
| --- | --- |
| Parameter | Keterangan |
| hProcess | Sebuah *handle* ke memori proses yang dimodifikasi. |
| lpBaseAddress | Penujuk ke alamat dasar dalam proses yang ditentukan kemana data ditulis. |
| lpBuffer | Penunjuk ke *buffer* yang berisi data yang akan ditulis di ruang alamat proses yang ditentukan. |
| nSize | Jumlah *byte* yang akan ditulis ke proses yang ditentukan. |
| lpNumberOfBytesWritten | Penunjuk ke variabel yang menerima jumlah *byte* yang di-*transfer* ke dalam proses tertentu. |

#### **Fungsi VirtualProtectEx**

Fungsi ini digunakan untuk merubah proteksi pada bagian dari *page* tertentu di *virtual address space* dari proses yang sedang dipanggil. Berikut sintaks C++ fungsi ini [15].

BOOL WINAPI VirtualProtectEx(

\_In\_  HANDLE hProcess,

\_In\_  LPVOID lpAddress,

\_In\_  SIZE\_T dwSize,

\_In\_  DWORD  flNewProtect,

\_Out\_ PDWORD lpflOldProtect

);

Parameter dari fungsi ini dapat dilihat pada Tabel 2.3.

**Tabel 2.3 Parameter Fungsi VirtualProtextEx**

|  |  |
| --- | --- |
| **Parameter** | **Keterangan** |
| hProcess | *Handle* untuk proses yang perlindungan memorinya harus diubah. |
| lpAddress | Penunjuk ke alamat dasar dari daerah halaman yang atribut proteksi aksesnya harus diubah. |
| dwSize | Ukuran wilayah yang atribut perlindungan aksesnya diubah, dalam *byte*. |
| flNewProtect | Opsi perlindungan memori. |
| lpflOldProtect | Penunjuk ke variabel yang menerima perlindungan akses sebelumnya dari halaman pertama di daerah halaman tertentu. |

#### **Fungsi VirtualQueryEx**

Fungsi ini digunakan untuk mengambil informasi mengenai berbagai halaman di *virtual address space* dari proses tertentu. Berikut sintaks C++ fungsi ini [16].

SIZE\_T WINAPI VirtualQueryEx(

\_In\_     HANDLE                    hProcess,

\_In\_opt\_ LPCVOID                   lpAddress,

\_Out\_    PMEMORY\_BASIC\_INFORMATION lpBuffer,

\_In\_     SIZE\_T                    dwLength

);

Parameter dan keterangan parameter yang dibutuhkan untuk fungsi ini dapat dilihat pada Tabel 2.4.

**Tabel 2.4 Parameter Fungsi VirtualQueryEx**

|  |  |
| --- | --- |
| **Parameter** | **Keterangan** |
| hProcess | *Handle* ke proses informasi memori. |
| lpAddress | *Pointer* ke alamat dasar dari daerah halaman. |
| lpBuffer | Penunjuk ke struktur MEMORY\_BASIC\_INFORMATION tempat informasi tentang rentang halaman tertentu dikembalikan. |
| dwLength | Ukuran *buffer* ditunjukkan oleh parameter lpBuffer, dalam *byte*. |

* 1. Windows *Debugger*

Windows merupakan sistem operasi yang menyediakan fasilitas yang sangat lengkap untuk melakukan *debugging*. *Debugger* menurut *Microsoft Developer Network* (MSDN) merupakan suatu aplikasi yang digunakan pengembang untuk mengamati program, memeriksa program, serta memperbaiki program yang salah ketika dijalankan atau saat *runtime* [17]. *Debugger* memiliki fungsi yang memungkinkan pengembang melakukan *stepping* atau langkah dari proses dan *thread* yang sedang dieksekusi. Fungsi lain dari *debugging* yaitu memantau register *processor*, memori, dan elemen lain dari konteks proses dan *thread*. Fasilitas yang disediakan windows tersebut sudah dapat digunakan untuk membuat *debugger* dasar dalam arti *debugger* mampu memberi notifikasi berupa informasi pada setiap *event* tertentu dalam proses yang sedang di-*debug*. Notifikasi ini memungkinkan *debugger* untuk mengambil tindakan yang sesuai dalam menanggapi *event* tersebut [18]. *Platform* popular untuk Windows *debugger* yang telah ada dan digunakan sebagai piranti lunak untuk menyingkapkan kerentanan dan pengembangan eksploit adalah Immunity Debugger dan OllyDbg. Keduanya merupakan sebuah *debugger* untuk 32-bit Windows. Akan tetapi, penelitian ini menganggap cukup sulit mengekstensi *debugger* tersebut karena sangat kompleks dan memiliki fitur-fitur yang tidak dibutuhkan seperti tampilan grafis. Oleh sebab itu, penelitian akan menggunakan *debugger* yang sederhana yng memiliki fungsi - fungsi dasar dan mengekstensinya.

Ada beberapa kelompok *debugger* yang diklasifikasikan berdasarkan informasi yang harus diperoleh saat melakukan *debugging* yaitu *white-box debugging* dan *black-box debugging*. *White-box debugging* adalah sebuah *debugger* yang dibuat sebagai *bit in functionality* yang ada di *Integrared* *Development* *Environment* (IDE). *Debugger* ini lazim digunakan untuk menelusuri kode sumber ketika terjadi *runtime* *error* atau untuk mencari dan menyingkapkan *logic error*. Sementara *black-box debugger* berlawanan dengan metode *white-box debugging* karena tidak memeiliki akses terhadap kode sumber. *Black-box debugger* mengoperasi program yang telah dikompilasi dan di *link* dengan cara, pertama sekali melakukan *disassembly* terhadap berkas *file* *executable.*

Ditinjau dari *privilege*, *black-box debugger* dibagi dua yaitu *user mode* (*ring* 3) dan kernel *mode* (ring 0). *User* *mode* merupakan sebuah *mode* dimana hak akses processor sama dengan user program. *Mode* ini tidak dapat secara langsung mengeksekusi perintah - perintah dengan *privilege* tinggi seperti *input* *output,* akan tetapi layanan sistem operasi diperoleh dengan melakukan *system* *call*. Sementara kernel *mode* merupakan sebuah *mode* dimana *core system* berjalan menggunakan hak atau *privilege* paling tinggi, contohnya menangkap paket dengan *network* *adapter* pada mode pasif [3].

* + 1. Windows *Debuging* API

Sistem operasi Windows menyediakan sekumpulan fungsi untuk men-*debug* *executable* yakni *debugging* API [19]. Terdapat beberapa fungsi *debugging* yang penting digunakan dalam membangun sebuah *debugger*, yaitu fungsi ContinueDebugEvent, DebugActiveProcess, DebugActiveProcessStop, dan WaitForDebugEvent. Penjelasan fungsi tersebut dijelaskan pada sub bab berikut.

* + - 1. Fungsi ContinueDebugEvent

Fungsi ini digunakan untuk mengaktifkan *debugger* untuk melanjutkan proses yang sebelumnya di-*debugging event*. Berikut sintaks C++ fungsi ini [20].

BOOL WINAPI ContinueDebugEvent(

\_In\_ DWORD dwProcessId,

\_In\_ DWORD dwThreadId,

\_In\_ DWORD dwContinueStatus

);

Parameter dan keterangan parameter yang dibutuhkan untuk fungsi ini dapat dilihat pada Tabel 2.5.

**Tabel 2.5 Parameter Fungsi ContinueDebugEvent**

|  |  |
| --- | --- |
| **Parameter** | **Keterangan** |
| dwProcessId | Identitas proses yang akan digunakan |
| dwThreadId | Identitas dari *thread* yang akan digunakan |
| dwContinueStatus | Opsi untuk melanjutkan *thread* yang akan digunakan di-*debugging event* |

* + - 1. Fungsi DebugActiveProcess

Fungsi ini digunakan untuk mengaktifkan *debugger* untuk mengambil proses yang aktif dan men-*debug* nya. Berikut sintaks C++ fungsi ini [21].

BOOL WINAPI DebugActiveProcess(

\_In\_ DWORD dwProcessId

);

Parameter dan keterangan parameter yang dibutuhkan untuk fungsi ini dapat dilihat pada Tabel 2.6.

**Tabel 2.6 Parameter FungsiDebugActiveProcess**

|  |  |
| --- | --- |
| **Parameter** | **Keterangan** |
| dwProcessId | Identitas proses dari proses yang akan digunakan |

* + - 1. Fungsi DebugActiveProcessStop

Fungsi ini digunakan ketika melakukan *detach* *process* dengan menghentikan *debugger* melakukan *debugging*. Berikut sintaks C++ fungsi ini [22].

BOOL WINAPI DebugActiveProcessStop(

\_In\_ DWORD dwProcessId

);

Parameter dari fungsi ini dapat dilihat pada Tabel 2.7.

**Tabel 2.7 Parameter Fungsi DebugActiveProcessStop**

|  |  |
| --- | --- |
| **Parameter** | **Keterangan** |
| dwProcessId | Identitas proses dari proses yang akan digunakan |

* + - 1. Fungsi WaitForDebugEvent

Fungsi digunakan untuk menunggu *debugging event* muncul pada suatu proses. Berikut sintaks C++ fungsi ini [23].

BOOL WINAPI WaitForDebugEvent(

\_Out\_ LPDEBUG\_EVENT lpDebugEvent,

\_In\_  DWORD         dwMilliseconds

);

Parameter dari fungsi ini dapat dilihat pada Tabel 2.8.

**Tabel 2.8 Parameter Fungsi WaitForDebugEvent**

|  |  |
| --- | --- |
| **Parameter** | **Keterangan** |
| lpDebugEvent | Pointer ke struktur DEBUG\_EVENT yang menerima informasi mengenai *debugging event*. |
| dwMilliseconds | Jumlah milidetik yang digunakan untuk menunggu *debugging event*. Jika parameter bernilai 0, tes fungsi untuk *debugging event*. Jika parameter *INFINITE*, fungsi tidak kembali hingga *debugging* *event* lain muncul. |

### **2.7.2 *Debug Event***

*Debug event* merupakan suatu peristiwa yang muncul ketika terjadi suatu *event* pada sistem seperti membuat proses, membuat thread, melakukan *load* DLL, melakukan *unload* DLL, membuat *output string* dan menghasilkan *exception*. Berikut struktur *debug event* dalam sintaks C++ [24].

typedef struct \_DEBUG\_EVENT {

DWORD dwDebugEventCode;

DWORD dwProcessId;

DWORD dwThreadId;

union {

EXCEPTION\_DEBUG\_INFO      Exception;

CREATE\_THREAD\_DEBUG\_INFO  CreateThread;

CREATE\_PROCESS\_DEBUG\_INFO CreateProcessInfo;

EXIT\_THREAD\_DEBUG\_INFO    ExitThread;

EXIT\_PROCESS\_DEBUG\_INFO   ExitProcess;

LOAD\_DLL\_DEBUG\_INFO       LoadDll;

UNLOAD\_DLL\_DEBUG\_INFO     UnloadDll;

OUTPUT\_DEBUG\_STRING\_INFO  DebugString;

RIP\_INFO                  RipInfo;

} u;

} DEBUG\_EVENT, \*LPDEBUG\_EVEN

**Tabel 2.9. Parameter Struktur DEBUG\_EVENT**

Parameter dan keterangan parameter yang dibutuhkan untuk fungsi ini dapat dilihat pada Tabel 2.9.

|  |  |
| --- | --- |
| **Parameter** | **Keterangan** |
| dwDebugEventCode | Kode yang mengidentifikasi tipe *debugging event* yang digunakan. |
| dwProcessId | Identitas proses pada *debugging event* yang muncul. |
| dwThreadId | Identitas *thread* pada *debugging event* yang muncul. |
| Union (Exception, CreateThread, CreateProcessInfo, ExitThread, ExitPorcess, loadDll, UnloadDll, DebugString, RipInfo) | Informasi tambahan mengenai *debugging event*. |

Menurut Windows *debugger*, ada beberapa tipe dari *debugging event* [25]:

* CREATE\_PROCESS\_DEBUG\_EVENT muncul sebelum menginisialisasikan proses baru yang akan di-*debug* atau pada saat yang sama *debugger* mengambil proses yang aktif.
* EXIT\_PROCESS\_DEBUG\_EVENT muncul ketika proses yang di-*debug* keluar atau berhenti.
* CREATE\_THREAD\_DEBUG\_EVENT muncul ketika proses yang sedang di-*debug* membuat *thread* baru.
* EXIT\_THREAD\_DEBUG\_EVENT muncul ketika *thread* pada proses yang sedang di-*debug* keluar atau berhenti.
* LOAD\_DLL\_DEBUG\_EVENT muncul ketika proses yang sedang di-*debug* me-*load* DLL (*Dynamically Linked Library*).
* UNLOAD\_DLL\_DEBUG\_EVENT muncul ketika proses yang sedang di-*debug* melepaskan DLL.
* EXCEPTION\_DEBUG\_EVENT terjadi ketika *exception* muncul pada proses yang sedang di-*debug*.
* OUTPUT\_DEBUG\_STRING\_EVENT muncul ketika proses yang sedang di-*debug* memanggil fungsi OutputDebugString.

Salah satu sifat *debugger* yaitu berjalan sebagai *loop* yang berulang-ulang untuk menunggu *debugging* *event* muncul. Ketika *debugging* *event* muncul, selanjutnya *loop* *break*, dan *event* *handler* yang diperlukan akan dipanggil. Ketika *event* *handler* dipanggil, *debugger* berhenti dan menunggu arahan untuk melanjutkannya kembali. Beberapa *event* umum yang dilakukan *debugger* yaitu menentukan *breakpoint*, mengatasi *memory* *violation* atau disebut pelanggaran akses atau *segmentation* *faults*, dan mengatasi *exception* yang dihasilkan saat program di-*debug*.

* 1. *Breakpoint*

*Breakpoint* merupakan suatu penanda yang digunakan *debugger* dalam menghentikan proses yang dianalisis untuk sementara waktu. Tujuan dari *breakpoint* adalah membuat program berhenti pada interval yang dijadwalkan sehingga status kode program dapat diperiksa secara bertahap pada saat *debugging*. Dengan menghentikan proses, kita dapat memeriksa variabel, *stack* *argument*, dan lokasi memori tanpa mengubah nilai dari proses itu sebelum direkam atau disimpan terlebih dahulu. Ada 3 tipe *breakpoint* yang utama yaitu *soft breakpoint*, *hardware breakpoint*, dan *memory* *breakpoint* [3]*.*

* + 1. *Software Breakpoint*

*Software* *breakpoint* dilakukan dengan mengganti sekumpulan instruksi dengan instruksi penghentian program, contohnya INT3. INT3 merupakan instruksi untuk menghentikan program, kode *byte* dari INT3 adalah 0xCC. Untuk dapat memahami cara kerja dari *software breakpoint*, perlu diketahui perbedaan instruksi dan *opcode* pada x86 *assembly*. Instruksi *assembly* adalah representasi *high-level* dari perintah yang dieksekusi CPU. Contohnya:

|  |
| --- |
| MOV EAX, EBX |

Sementara *operation code* atau *opcode*, merupakan *command* bahasa mesin yang dieksekusi CPU. Contohnya, perubahan dari instruksi diatas kedalan *opcode*:

|  |
| --- |
| 8BC3 |

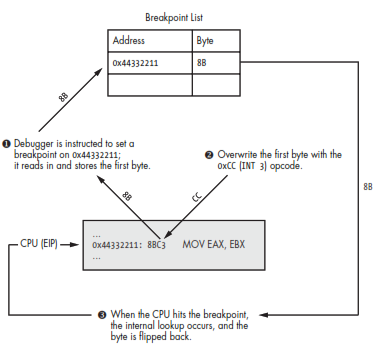
Jika instruksi diatas berada di alamat 0x44552211, representasinya akan terlihat demikian:

|  |
| --- |
| 0x44332211: **8B**C3 MOV EAX, EBX |

Hasil perubahan *opcode* setelah di *set* *breakpoint*-nya.

|  |
| --- |
| 0x44332211: **CC**C3 MOV EAX, EBX |

Dapat dilihat bahwa instruksi telah melakukan *swap byte* 8B dan digantikan dengan *byte* CC. Ketika CPU menemukan *byte* tersebut, CPU berhenti dan memunculkan *event* INT3. Selanjutnya *debugger* menentukan titik penghentian program atau *set* *breakpoint* pada alamat tertentu, maka akan dibaca *byte opcode* pertama pada alamat yang diminta dan akan disimpan. Kemudian *debugger* menulis *byte* CC ke alamat tersebut. Ketika *breakpoint* atau INT3 diberikan, *event* yang di-*trigger* oleh CPU yaitu menginterpretasikan *opcode* CC, lalu *debugger* akan menangkapnya. *Debugger* kemudian memeriksa penunjuk atau *pointer* ke instruksi (*register* EIP) yang menunjuk ke alamat yang telah di-*set breakpoint* sebelumnya. Jika alamat ditemukan pada daftar *breakpoint* milik *debugger*, *debugger* akan menulis kembali *byte* yang disimpan ke alamat tersebut agar *opcode* dapat dieksekusi dengan baik setelah proses dilanjutkan. Proses *soft breakpoint* digambarkan pada Gambar 2.3.



**Gambar 2.3 Konsep Software Breakpoint**

* + 1. *Hardware Breakpoint*

*Hardware breakpoint* digunakan ketika jumlah *breakpoint* yang dibutuhkan bernilai kecil dan perangkat lunak yang di-*debug* tidak dapat dimodifikasi. *Breakpoint* jenis ini biasanya menggunakan *debug register* (DR). Caranya dengan memodifikasi nilai-nilai *debug register* dari 0 hingga *debug register* 7 dengan operasi-operasi *bit masking*. Adapun 8 *register* (register DR0 hingga DR7), yang digunakan untuk men-*set* dan mengatur *hardware breakpoint*. *Debug* *register* DR0 hingga DR3 digunakan untuk alamat-alamat *breakpoint* sehingga kita dapat memakai hanya sampai 4 *hardware breakpoint* pada saat yang bersamaan. *Register* DR4 hingga DR5 sudah digunakan, dan DR6 digunakan untuk status *register*, yang menentukan tipe *debugging event* yang di-*trigger* saat *breakpoint* ditemukan. *Debug register* DR7 merupakan bagian penting dari *hardware breakpoint* berisi *switch on/off* bagi *hardware breakpoint* dan juga menyimpan kondisi *breakpoint*. Dengan mengatur *flag* tertentu di *register* DR7, kita dapat membuat *breakpoint* dengan kondisi berikut:

* *Break* ketika instruksi di eksekusi pada alamat tertentu.
* *Break* ketika data ditulis ke sebuah alamat.
* *Break* untuk membaca atau menulis ke alamat tertentu namun tidak di eksekusi.
  + 1. *Memory Breakpoint*

*Memory breakpoint* dilakukan dengan menentukan hak-hak akses pada suatu alamat memori sehingga ketika program itu berjalan, *memory breakpoint* akan berusaha mengakses memori yang sebenarnya tidak bisa diaksesnya, disebut dengan *access violation*. *Access violation* akan ditangkap, kemudian di-*set* level hak akses memori lebih tinggi dari hak akses yang harusnya bisa dilakukan setelah itu akan dikembalikan hak akses level nya ke level yang bisa dibaca program. Ketika *debugger* mengatur *memory breakpoint*, maka akan mengubah izin pada wilayah atau *page* dari memori. *Memory* *page* adalah bagian terkecil dari memori yang mengoperasikan *handle* sistem. *Memory page* juga harus memiliki izin akses agar dapat dialokasikan. Beberapa contoh izin *memory page* adalah sebagai berikut:

* Eksekusi *page* merupakan izin untuk melakukan eksekusi meskipun hal tersebut menjadi pelanggaran akses ketika proses membaca atau menulis ke *page*.
* Membaca *page* merupakan izin proses untuk membaca dari *page*.
* Menulis *page* merupakan izin proses untuk ditulis ke *page*.
* *Guard* *page* adalah suatu *page* yang di-*set* oleh *debugger* dengan hak akses yang lebih tinggi dari *user* program sehingga ketika *page* tersebut diakses oleh piranti lunak yang sedang dianalisis akan terjadi *exception*. *Exception* ini akan ditangani oleh *debugger* dan untuk melanjutkan eksekusi *guard* *page* akan dikembalikan *privilege*-nya dengan *privilege* yang sama dengan *user* program.
  1. Konsep *Memory Scanning*

Manajemen memori pada Microsoft Windows NT dibangun pada fungsi yang sediakan *hardware*. Salah satu yang paling umum adalah konsep manajemen memori dari arsitektur Intel CPU IA-32. Menurut arsitektur ini memori diatur dalam *page* berukuran 4096*byte* dan4MB [26].

*Memory scanning* dikelompokkan atas dua *mode* yaitu *user-mode* dan *kernel-mode*. Penelitian ini fokus kepada *user* *mode*. Pada *user-mode*, cara mengakses data proses tertentu di memori adalah melalui API yang disebut ReadProcessMemory(). API ini digunakan oleh *debugger* untuk mengatur eksekusi dari program aplikasi yang dianalisis oleh *debugger*. API ini juga menggunakan *handle* proses yang dapat diperoleh dari API OpenProcess() dan akses PROCESS\_VM\_READ.

* 1. *Malware Signature*

*Malware signature* merupakan sebuah tanda atau *signature* unik yang struktur penandanya dirancang tidak sama satu dengan lainnya. Signature ini berupa sekumpulan *byte* atau *opcode* yang menerangkan bahwa *executable* merupakan *malware*. Ide penelitian adalah diekstensinya sebuah *debugger* yang telah menerapkan algoritma penentuan titik waktu pengambilan memori dimana ketika titik tersebut dicapai kode-kode deofkuskasi telah selesai dieksekusi dengan cara demikian isi memori proses diharapkan telah memiliki tanda-tanda atau *signature* piranti lunak jahat. Oleh sebab itu, penelitian ini membutuhkan *malware signature*. Kumpulan malware signature yang digunakan berasal dari basis data ClamAV [27].

* + 1. Struktur *Signature*

Pada bagian ini didentifikasi beberapa properti umum dari *signature* yaitu [27]:

* Nama. *Signature* harus memiliki nama sehingga ketika dilakukan pengecekan pada antivirus muncul notifikasi nama *malware* yang teridentifikasi. Salah satu konvensi penamaan *malware* yaitu CARO (*Computer Antivirus Researchers Organization*). Penamaan *malware* ini ditemukan oleh anggota CARO pada tahun 1991 dan diterima oleh dunia pada tahun 2002. Format umum dari penamaan *malware* dengan standar CARO adalah sebagai berikut:

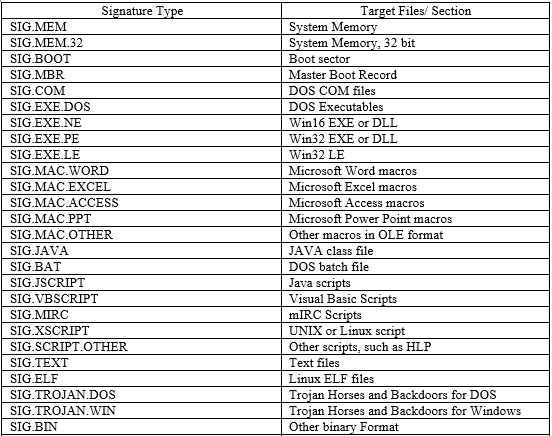
[<type>://][<platform>/]<family>[.<group>]

[.<length>].<variant>[<modifiers>][!<comment>]

Sebagai contoh “virus://{W97M,X97M,PP97M}/Tristate.A” dikenal sebagai virus “Tristate.A” pada *end user*.

* Tipe. *Signature* diklasifikasikan kedalam tipe infeksi yang dilakukannya seperti *boot sector*, COM *file*. Pengklasifikasian tipe *signature* pada Ikarus *Software* dapat dilihat pada Tabel 2.10.
* *Bytepattern*. *Bytepattern* merupakan bagian penting dari *signature* karena *bytepattern* merupakan urutan kode yang mengidentifikasi *malware*.

# **Tabel 2.10 Tipe *Signature*** [27]



* + 1. CVD (ClamAV *Virus Database*)

CVD merupakan *signature database* dalam berbagai format. *Signature* yang digunakan adalah ClamAV *signature test file*. Letak *signature* dari *malware* ada pada bagian *header* ASCII ditiap *file* CVD. Terdapat dua *database* CVD pada ClamAV yaitu main.cvd (berisi *signature* statis) dan daily.cvd (berisi *signature* dengan *regex* sederhana). Ada beberapa format *signature* pada ClamAV, yaitu [28]:

1. *Hash-based signatures*

Pada format ini, menggunakan *checksum file hash* dan biasanya digunakan untuk *malware* statis. Metode yang digunakan seperti: MD5 (.hdb), SHA1 dan SHA256 (.hsb), PE *section hash signature (.mdb). Hash-based signature* tidak dapat digunakan pada *text file, HTML* dan data yang di-*preprocessing* sebelum *pattern* dicocokkan*.*

1. *Body-based signatures*

*Signature* pada format ini disimpan dalamformat *hexadecimal.* Berikut adalah contoh *signature malware* dalam *hexadecimal*.

Beberapa informasi penting pada *body-based signature* yaituformat dasar *file* yang berisi informasi *signature* adalah .db. Ada juga format *signature* tambahan yang menampung informasi tambahan seperti tipe *file* target, *offset* virus atau versi mesin. Informasi tambahan tersebut membantu pendeteksian virus semakin baik. Sebagai contoh format dari *signature*:

NamaMalware:TipeTarget:Offset:HexSignature[:MinFL:[MaxFL]]

Dimana TipeTarget dispesifikasikan kedalam beberapa tipe target *file*:

* 0 = semua jenis *file*
* 1 = *Portable Executable*, baik 32-bit dan 64-bit
* 2 = container OLE2, termasuk macro spesifiknya. Format OLE2 biasanya digunakan pada *file* instalasi MS Office dan MSI
* 3 = HTML, Javascript
* 4 = *file* pesan
* 5 = grafik
* 6 = ELF
* 7 = *file* teks ASCII (yang telah dinormalisasi)
* 8 = *File* yang tidak digunakan
* 9 = *File* Mach-O
* 10 = *File* PDF
* 11 = *File flash*
* 12 = *File class* Java

Untuk *offset* yaitu simbol *asterisk* atau bilangan desimal n yang mungkin dikombinasikan dengan pengubah tertentu:

1. \* = semua jenis *offset*
2. n = *offset* absolut
3. EOF-n = akhir *file* dikurangi n *byte*

Parameter MinFL (*The minimum* ClamAV *feature* *level*) dan MaxFL (*The maximum* ClamAV *feature* *level*) bersifat opsional, parameter ini digunakan untuk membatasi *signature* berdasarkan waktu periode minimum dan maksimum suatu *engine* tertentu *release*. Semua *signature* yang berisi informasi *malware* harus ditempatkan pada *file* .ndb. Sementara untuk *logical signature* disimpan dalam format *file* .ldb.

1. *Icon signatures for PE files (.idb)*

Format dari *file icon signature* untuk PE *File* yaitu:

ICONNAME: GROUP1: GROUP2: ICON\_HASH

Dimana:

1. ICONNAME merupakan *string* unik yang mengidentifikasi spesifik ikon.
2. GROUP1 merupakan identifikasi *string* pada grup pertama dari ikon.
3. GROUP2 merupakan identifikasi *string* pada grup kedua dari ikon.
4. ICON\_HASH merupakan *hashing fuzzy* pada ikon gambar.
   1. *Multi-Pattern String Matching*

Pada *multi-pattern string matching* merupakan upaya untuk mencari pola (dalam hal ini sekumpulan *string*) pada suatu teks/*string*. Adapun beberapa penerapan algoritma yang dapat digunakan untuk *string matching.* [29]

* + 1. Algoritma Boyer-Moore

Algoritma Boyer-Moore diperkenalkan oleh Bob Boyer dan J.S. Moore pada tahun 1977. Pada algoritma ini pencocokan kata dimulai dari karakter terakhir kata kunci menuju karakter awalnya. Jika terjadi perbedaan antara karakter terakhir kata kunci dengan kata yang dicocokkan maka karakter-karakter dalam potongan kata yang dicocokkan tadi akan diperiksa satu per satu. Hal ini dimaksudkan untuk mendeteksi apakah ada karakter dalam potongan kata tersebut yang sama dengan karakter yang ada pada kata kunci. Apabila terdapat kesamaan, maka kata kunci akan digeser sedemikian rupa sehingga posisi karakter yang sama terletak sejajar, dan kemudian dilakukan kembali pencocokan karakter terakhir dari kata kunci. Sebaliknya jika tidak terdapat kesamaan karakter, maka seluruh karakter kata kunci akan bergeser ke kanan sebanyak m karakter, di mana m adalah panjang karakter dari kata kunci [30].

Ada 3 pendekatan yang digunakan pada algoritma ini yaitu *scan* dari kanan ke kiri, aturan *bad character* dan aturan *good suffix shift*.

1. *Scan* dari kanan ke kiri

Pendekatan ini menerapkan pemindaian dari kanan ke kiri pada deretan pola P yang dicari untuk diperiksa kecocokannya pada teks T.

1. Aturan *bad character*

Pendekatan ini diterapkan ketika karakter pada T bertemu kondisi x ≠ y dimana x sebagai karakter di alpabet dan y sebagai karakter terakhir dari P. Ketika kecocokan muncul, cari posisi P berkarakter x, lalu geser P kekanan hingga bertemu x pada P tepat dibawah ketidakcocokan x pada T. Namun apabila x tidak muncul pada P, kita dapat menggeser P hingga melewati poin dimana ketidakcocokan T terjadi. R(x) sebagai posisi kemunculan karakter X pada P. R(x) didefinisikan 0 apabila x tidak ada pada P.

1. Aturan *good* *shift*

Pendekatan ini terjadi ketika terdapat *substring* t pada T yang cocok dengan *suffix* atau *substring* dari P. Selanjutnya geser P hingga kondisi berikut terpenuhi yaitu kemunculan dari t pada P cocok dengan t pada T, prefiks dari P cocok dengan *suffix* dari t, dan P digeser melewati t. Algoritma Boyer-Moore mencari *pattern* dengan kompleksitas waktu sebesar .

* + 1. Algoritma Rabin-Karp

Algoritma Rabin Karp adalah algoritma pencarian *string* yang ditemukan oleh Michael O. Rabin dan Richard M. Karp, algoritma ini melakukan pencarian *string* didalam teks dengan memanfaatkan *hash function.* Algoritma ini kurang baik apabila diterapkan dalam pencarian *single pattern*. Akan tetapi akan sangat penting dan sangat efektif pada pencarian *multiple pattern*. Fungsi *hash* adalah sebuah fungsi yang mengkonversi setiap *string* menjadi bilangan, yang disebut *hash value*. Pada algoritma ini *string* yang akan dicari diubah menjadi bilangan yang biasa disebut sebagai *hash value*, sebagai contoh jika ada *string* “*hello*” maka *hash value* dari *string* tersebut adalah 5. Algoritma ini berprinsip bahwa jika dua buah *string* sama maka *hash value* dari kedua *string* tersebut pasti sama, sehingga algoritma hanya akan mencari *string* yang memiliki *hash value* yang sama dengan *value* dari *string* yang dicari.

Algoritma ini memiliki kelemahan yang disebabkan oleh prinsipnya dimana semua *string* yang ingin dicari dianggap pasti memiliki *hash value* yang sama dengan *string* yang didapat dan algoritma hanya membandingkan *hash value* keduanya maka akan sering terjadi kasus di mana *string* yang didapat bukanlah *string* yang dicari namun memiliki *hash value* yang sama, contohnya “*hello*” dan “katak”.

Persamaannya adalah sebagai berikut:

( 2.1 )

Dengan H adalah nilai *hash*, C1 adalah nilai ASCII suatu karakter, a adalah basis, k adalah banyaknya karakter.

Ada banyak cara dalam mengkonversi *string* menjadi *hash value*, salah satunya adalah *Rolling Hash Function*, dimana *hash* mengkonversikan *string* menjadi bilangan dengan basis bilangan prima yang besar untuk menentukan modulo. Bilangan prima yang besar menghasilkan representasi angka hasil *hashing* pola kata agar tidak terjadi *quadratic* (hasil *hashing* yang sama dengan pola kata yang berbeda). Semakin besar bilangan prima yang dipakai maka semakin kecil pula kesempatan terjadinya *quadratic* atau kesamaan nilai *hash* pada pola kata yang berbeda, waktu atau durasi pencocokan *string* mewakili *time complexity* dari algoritma Rabin-Karp dalam melakukan pencocokan *string*.

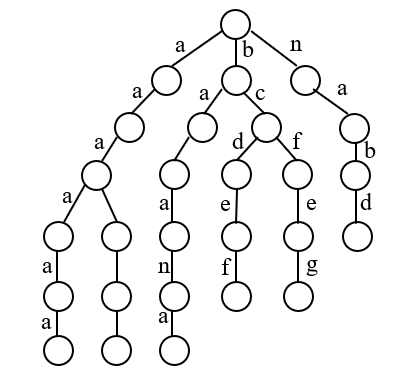
Tahapan pemindaian pada algoritma ini dilakukan secara iterasi, dimana tahapannya adalah sebagai berikut:

1. Kalkulasi nilai *hash* h berdasarkan karakter B dari teks.
2. Periksa nilai dari SHIFT[h]: jika > 0, geser teks dan kembali ke poin 1; jika sebaliknya, lanjut ke poin 3.
3. Kalkulasi nilai *hash* pada prefix teks (mulai dari karakter m ke kiri dari posisi sekarang); dan simpan sebagai teks\_prefiks.
4. Periksa pada tiap P, HASH[h] ≤ P ≤ HASH[h+1] apakah PREFIKS[P] = teks\_prefiks. Ketika sama, periksa pola sebenarnya terhadap teks secara langsung.

Algoritma Rabin-Karp mencari *pattern* dengan kompleksitas waktu sebesar . [31]

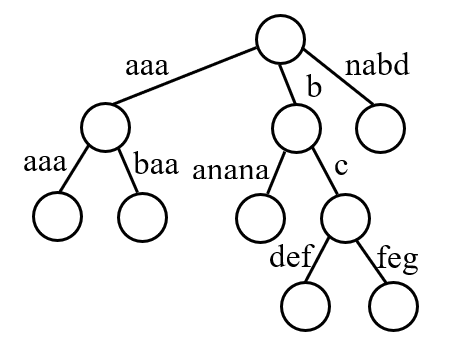
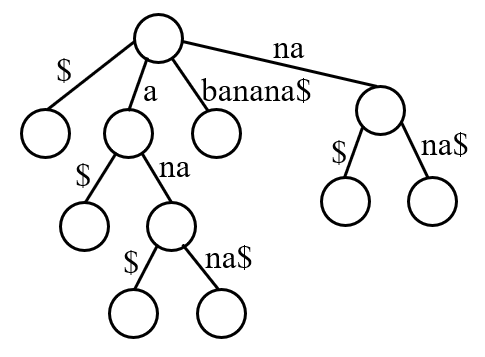
* + 1. Struktur Data Suffix Tree

*S uffix tree* merupakan suatu struktur data yang biasa digunakan pada pencocokan pola, pencarian *string* yang berbeda pada suatu *string* tertentu. *Suffix* *tree* adalah *compressed tree* dari semua *suffix* suatu *string*. Ilustrasi *compressed tree* akan dijelaskan dari penjelasan berikut. Pertama mari amati *tree* umum yang dapat dibentuk dari *set string* berikut ini: {“banana”, “nabd”, “bcdef”, “bcfeg”, “aaaaaa”, “aaabaa”} pada Gambar 2.4.



Pertama sekali ketika membangun *tree*, nilai karakter pertama suatu *set string* yang ingin dibangun struktur *tree*-nya disimpan didalam *node*. Masing-masing *node* yang dibentuk mulai dari *root* memiliki nilai karakter yang berbeda. Caranya dengan memeriksa apakah telah ada *node* yang menyimpan karakter yang berisi karakter dari *string* yang ingin dibangun struktur *tree*-nya. Jika tidak ditemukan karakter yang sama, maka bentuk *node* baru untuk menyimpan string tersebut. Jika ditemukan karakter yang sama, mulai dari dari *node* tersebut dan lakukan iterasi pengecekan karakter untuk membentuk *edge tree* yang baru. *Compressed tree* yaitu kondisi *edge* pada *tree* yang memiliki satu *child* akan digabungkan menjadi satu *edge* dan label *edge* akan dirangkai. Hasil *compressed tree* dari set *string* digambarkan pada Gambar *2.5*.

**Gambar 2.4 Tree Set String [39]**

Berdasarkan ilustrasi ditunjukkan bahwa setiap *node* dalam *compressed tree* setidaknya akan memiliki dua *child* serta memiliki N *leaf*, dimana N merupakan jumlah *string* yang disisipkan dalam *compressed tree* serta karakter pertama dari pola yang dicari tersebut akan dijadikan sebagai *root*. Terdapat dua hal yang perlu diketahui yaitu masing-masing *node* internal setidaknya memiliki dua *child* dan terdapat N *leaf* yang mengimplikasikan bahwa terdapat setidaknya 2N-1 *node* dalam *tree*. Maka dari itu, kompleksitas ruang *compressed tree* adalah O(N). Sering kali kita menambahkan karakter khusus yang tidak dimiliki bagian lain pada *string*. Misalnya, kita menambahkan $ pada *string*. Sebagai contoh *string* “banana” sehingga *string* yang baru adalah “banana$”. Kini *suffix tree* akan terlihat seperti Gambar 2.6.

**Gambar 2.5 Compressed Tree [39]**

**Gambar 2.6 *Suffix Tree* [37]**

*Suffix tree* biasanya digunakan dalam aplikasi *bioinformatics*, digunakan untuk mencari pola pada rangkaian DNA dan protein, yang bisa dilihat sebagai suatu *string* panjang. Kemampuan untuk menemukan secara efisien dengan adanya ketidak-cocokan teks mungkin adalah kekuatan terbesar *suffix tree*. [32]

## **Antivirus**

Antivirus merupakan program keamanan yang di-*install* pada komputer atau *mobile phone* yang digunakan untuk melindungi *device* dari infeksi *malware*. *Malware* yang dimaksud dapat berupa virus, *worm*, trojan dan *spyware*. Jika komputer terinfeksi oleh *malware*, maka *attacker* dapat mengakses dan memanipulasi data atau sistem yang ada pada komputer yang terinfeksi. Sehingga antivirus ini diciptakan agar komputer atau *device* kita aman dan tidak terinfeksi *malware*.

Ada dua cara yang digunakan antivirus dalam mengidentifikasi *malware*, yaitu dengan mendeteksi *signature malware* dan dari *behavior* atau perlakuan *malware* tersebut [33]

*Software* antivirus merupakan salah satu aplikasi yang rumit yang sistem aplikasinya harus menghadapi beragam jenis virus. Virus-virus tersebut dapat ditemukan di banyak tipe *file* dan format, yaitu: [34]

* *Executable* (exe, dll, msi, com, pif, cpl, elf, ocx, sys, scr, dsb).
* Dokumen (doc, xls, ppt, pdf, rtf, chm, hlp, dsb).
* *Compressed archive* (arj, arc, cab, tar, zip, rar, z, zoo, lha, lzh, ace, iso, dsb).
* *Executable packer* (ipx, fsg, mew, nspack, wwpack, aspack, dsb).
* *File* media (jpg, gif, swf, mp3, rm, wmv, avi, wmf, dsb).

Antivirus menjadi kebutuhan sistem untuk melindungi komputer dari potensi serangan virus maupun malware, perlindungan ini akan menjaga sistem dan informasi pribadi tidak tersebar melalui jaringan. Beberapa contoh antivirus yang tersedia tidak berbayar yaitu Norton AntiVirus, Avast, AVG Antivirus, Malwarebytes, dsb.

# **ANALISIS**

# Pada bab ini dijelaskan mengenai analisis yang dilakukan untuk pembangunan sistem pendeteksi *malware* dengan menggunakan *memory scanning*.

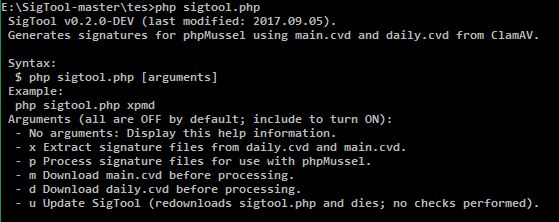
## **Analisis Data *Signature Malware***

*Malware* memiliki identitas unik yang diciptakan oleh penciptanya berupa *shellcode* *signature* dimana pada setiap *signature* di *malware* berbeda satu sama lain. Pada penelitian ini *resource* yang digunakan berupa *shellcode* *signature malware*. Fakta tersebut mendorong peneliti untuk menganalisis *shellcode* *signature* dari *malware* dan menggunakannya dalam proses pendeteksian *file executable* dengan sistem *custom debugger* yang akan dibangun.

Cara untuk mendapatkan *signature malware* dari program yang diindikasi mengandung *malware* sulit dilakukan khususnya apabila *malware* menggunakan teknik-teknik penghindaran *debugger* seperti melakukan *packed file*, melakukan obfuskasi, menerapkan teknologi *anti*-*sandbox*, serta melakukan *shellcode encoding*. *Shellcode encoding* merupakan teknik transformasi kode yang bertujuan untuk mengubah sintaks dari program tanpa mengubah fungsi program tersebut. *Shellcode encoding* dibutuhkan untuk menghindari karakter jahat, seperti \x00, \x0a, \x0d, \xff dst, dan digunakan untuk menghindari pendeteksian *antivirus*. Komponen yang dibutuhkan dalam melakukan *encode shellcode* ialah *encoder* dan *decoder*. *Encoder* berfungsi untuk mengganti kode sintaks dan *decoder* berfungsi untuk merubah kode yang telah di-*encode* kembali kebentuk aslinya.

*File executable* dalam bentuk format *compressed* disebut sebagai *packed file*. Untuk dapat mengatasi *packed file*, harus melakukan *decompress* atau *unpacked file* tersebut. Apabila suatu program dienkripsi menggunakan *packer* maka data heksadesimal milik program juga akan berubah.

Kumpulan *signature* *malware* yang akan digunakan pada penelitian ini berasal dari *library* ClamAV berupa *file* dengan format CVD (ClamAV Virus *Database*) dengan nama main.cvd yang dapat diperoleh dari *website* ClamAV. Sebelum dapat menggunakan *library* ini peneliti terlebih dahulu harus melakukan *unpack* terhadap *library* menggunakan *command* *sigtool* dari ClamAV untuk memperoleh informasi *malware* didalam *library* tersebut seperti pada Gambar 3.1.

**Gambar 3.1 Sigtool Command**

Hasil *unpack* dari *command* tersebut adalah *file* berisi *malware signature* dengan format .ndb yang berisi 100566 *signature.* Apabila format *malware* diidentifikasi setiap *signature* *malware* terdiri dari nama *malware*, tipe target, *offset* dan *signature* berupa heksadesimal. Sebagai contoh dapat dilihat pada *signature malware* contoh dibawah ini.

Adapun hasil identifikasi *malware,* diperoleh informasi yang dapat dilihat pada Tabel 3.1.

# **Tabel 3.1 Hasil Identifikasi *Malware***

|  |  |
| --- | --- |
| **Nama *malware*** | Trojan dengan nama DustySky-22 berbasis Windows |
| **Tipe target** | *Portable Executable* |
| **Jenis *offset*** | Semua jenis *offset* |
| ***Signature* heksadesimal** | 4175746f44656331396669782e657865 |

## **Windows *Custom* *Debugger***

Pada analisis *custom debugger* ini, peneliti mengembangkan *custom debugger* untuk men-*debug* *file executable* yang akan memeriksa apakah *file* tersebut mengandung *malware* atau tidak. Dalam pembuatan *custom debugger* ini, peneliti mengimplementasikan kode Python yang ada pada buku “Gray Hat Python” dengan penulis Justin Seitz. Windows *custom debugger* yang dikembangkan oleh Justin Seitz merupakan adaptasi dari Immunity Debugger [3]. Peneliti akan meng-*extend debugger* yang dikembangkan oleh Justin Seitz untuk melakukan *scan* memori proses yang sedang berjalan untuk mengidentifikasi apakah perangkat lunak tersebut jahat atau tidak.

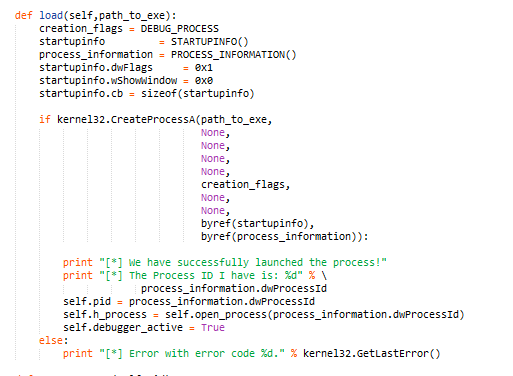
Sistem *custom debugger* akan menggunakan *file executable* sebagai obyek yang akan diperiksa*.* Kemudian *file executable* akan di-*debug* untuk membuka semua data yang ada pada *file* tersebut sehingga isi memori dapat diakses. *Custom debugger* mampu membuka *file executable*, menjalankannya atau menyisipkannya ke dalam proses yang berjalan. *Custom debugger* yang dibangun menggunakan Windows *debugging* API, yang menyediakan fungsi untuk mengembangkan *debugger*.

Pada awalnya, disediakan proses untuk mengakses *file* *executable*. Ada 2 cara yang digunakan yaitu *open process* dan *attach process*. Cara pertama dilakukan dengan memberikan kontrol penuh terhadap proses sebelum kode program dijalankan. Metode *open proces*s dilakukan dengan menjalankan *executable* dari *debugger* secara langsung. Untuk membuat proses, sistem akan memanggil fungsi CreateProcess(). Setelah membuat proses, peneliti menginput *path* *file executable* tersebut. Proses sudah dapat mengakses *file executable*. Cara kedua dilakukan dengan menghentikan proses sementara lalu mengambil program yang akan dianalisa. Dalam hal ini sistem akan memanggil fungsi OpenProcess() agar dapat mengakses proses sebelum men-*debug* *executable*. Pada cara kedua, peneliti menyisipkan PID dari *file executable*, contoh obyek *file* yang dianalisis yaitu calc.exe. *File* tersebut di jalankan secara manual, lalu PID proses nya akan diperoleh dari informasi yang diakses melalui *task manager*. PID tersebut akan dianalisis programnya dengan mengakses dan menganalisis isi memorinya. *Class debugger* pada sistem ini disediakan oleh kernel32 yang merupakan *library* Windows yang sudah ada.

Namun pada *custom debugger* ini cara untuk mendapatkan *alamat text section* yang baik adalah dengan cara pertama yaitu *open process*. *Open process* mampu menghentikan instruksi sementara setelah program dijalankan sehingga alamat dari *text section* dapat diperoleh. Sedangkan cara kedua alamat *text section* sulit ditemukan oleh *custom debugger* dikarenakan program telah berjalan sehingga tidak diketahui tepatnya dimana alamat yang akan diteliti.

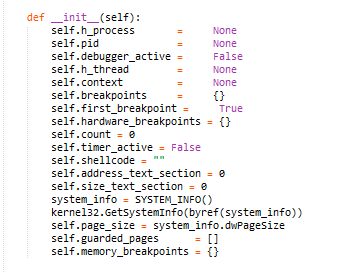
Ada beberapa fungsi *custom debugger* yang kami gunakan dan ada juga fungsi pada *custom debugger* yang kami modifikasi sesuai dengan kebutuhan penelitian ini. Fungsi – fungsi tersebut dapat dilihat pada kode program dibawah ini.

* Fungsi *load* yang digunakan untuk mengakses *file executable* yang akan diperiksa. Fungsi ini menggunakan fungsi kernel32 yaitu createProcess(). Kode program pada fungsi ini, dapat dilihat pada Gambar 3.2.



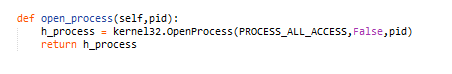
**Gambar 3.2 Kode Program Load Custom Debugger**

* Fungsi yang digunakan untuk mendefenisikan variable-variabel yang digunakan pada *custom debugger* adalah \_init\_. Kode program untuk fungsi ini dapat dilihat pada Gambar 3.3 dihalaman berikutnya.



**Gambar 3.3 Kode Program Inisialisasi Variable**

* Fungsi open\_process() dengan parameter *self* dan pid. Fungsi ini digunakan untuk mengakses proses sebelum men-*debug* *executable*. Kode program pada fungsi ini, dapat dilihat pada Gambar 3.4.



# **Gambar 3.4 Kode Program Open\_Process *Custom Debugger***

* Fungsi run() yang digunakan untuk mengeksekusi kode program pada *debugger*. Kode program pada fungsi ini dapat dilihat pada Gambar 3.5.

# **Gambar 3.5 Kode Program *run() Custom Debugger***

* 1. Penentuan Titik Penghentian Program atau *Breakpoint*

Penentuan titik penghentian program atau *breakpoint* digunakan untuk menentukan letak *breakpoint* sehingga dapat dipastikan semua kode-kode obfuskasi sudah terbuka. Tipe *breakpoint* yang digunakan di *custom* *debugger* ini adalah *soft breakpoint*, *breakpoint* tipe ini memiliki kemampuan menganalisis isi memori serta mengubah isi memori. Pertama, d*ebugger* men-*set* *breakpoint* menggunakan fungsi VirtualProtextEx untuk meminta izin melakukan perubahan *page* kode dari *read-execute* ke *read-write-execute*. Ketika *debugger* telah memiliki izin tersebut, kemudian sistem menggunakan fungsi ReadProcessMemory() untuk membaca dan menyimpan seluruh instruksi sebelumnya dari alamat memori yang akan diberi *breakpoint*. Selanjutnya untuk dapat menulis “INT3 opcode” pada alamat instruksi yang diberi *breakpoint* maka *debugger* akan menggunakan fungsi WriteProcessMemory(). Ruang lingkup dari *memory scanning* yaitu seluruh kode program yang akan diteliti dan ruang lingkup memori yang dibaca yaitu *text section.*

Berdasarkan analisis yang telah dilakukan pada penelitian ini, telah diidentifikasi beberapa teknik untuk menentukan teknik penghentian program.

* Teknik naif, teknik ini dinamakan naif karena cara kerja *debugger* yang menentukan *breakpoint* pada tiap instruksi dalam *text section*, jika terdapat 100 instruksi maka terdapat 100 *breakpoint* juga. Setiap *breakpoint* terjadi, hasil dari tiap instruksi akan disimpan. Instruksi akan berulang diambil hingga instruksi terakhir selesai. Metode ini memberikan hasil yang maksimum namun memiliki kelemahannya yaitu membutuhkan penyimpanan memori yang besar serta waktu yang dilakukan lama.
* Teknik acak, cara kerja teknik ini yaitu mengambil satu alamat pada *text section* secara acak dan men-*set* nya sebagai titik penghentian program.
* Teknikmen*-set breakpoint* pada instruksi terakhir *text section*

Teknik ini relatif mudah untuk mengambil instruksi terakhir pada *text section.* Kelemahan dari teknik ini ialah belum tentu instruksi terakhir yang akan terakhir dieksekusi atau instruksi terakhir ini yang membuka semua metode obfuskasi yang ada di *malware*.

* Teknik *Timer*

Pada metode ini, *breakpoint* akan di set berdasarkan waktu yang telah ditentukan. Pertama sekali akan di *set start time* pada alamat awal dari *text section* (*address of entry point*) dan waktu tersebut akan tetap berjalan sampai akhir dari alamat *text* *section*. Jika telah mencapai alamat terakhir dari *text section* maka *timer* akan diberhentikan dan akan diambil isi memori atau *snapshot* dari *text section* dari *address of entry point* sampai waktu telah berhenti.

## **Penentuan Teknik untuk Menghentikan *Debugger* Sementara**

*Debugger* akan dihentikan ketika waktu mengeksekusi memory telah selesai. Pertama sekali, peneliti menggunakan debug\_active\_process\_stop untuk menghentikan *debugger*. Namun setelah dianalisis, meskipun *debugger* berhenti peneliti tidak bisa mengambil *snapshot* memori tersebut dikarenakan fungsi dari debug\_active\_process\_stop adalah menghentikan *debugger* sekaligus menghancurkan memori tersebut. Selanjutnya, peneliti juga mencoba menggunakan fungsi detach(). Fungsi detach() merupakan salah satu fungsi untuk menyelesaikan proses *debugging* pada *debugger*, tetapi jika menggunakan proses detach() ini maka proses selanjutnya tidak akan dijalankan yang berarti proses *debugging* berhenti total.

Berdasarkan analisis tersebut, peneliti mendefenisikan fungsi *suspend* yang berisi status parameter debug\_active\_process. Fungsi debug\_active\_process() ini juga merupakan salah satu cara untuk menghentikan parameter PID sehingga *debugger* otomatis di-*suspend* atau diberhentikan sementara. Hal ini dapat terjadi jika status debug\_active\_process = false. Melalui cara ini, *debugger* dapat diberhentikan ketika selesai mengeksekusi memori sehingga peneliti dapat mengambil *snapshot* memori dan menggunakannya pada proses selanjutnya.

* 1. *Debug Event*

*Debugging event* merupakan kejadian pada sebuah proses yang sedang di-*debug* yang menyebabkan sistem memberikan notifikasi ke *debugger*. Peran *debugging event* pada *debugger* terjadi apabila telah memiliki hak akses ke proses target. Ketika akses terhadap proses target telah ada maka *handle debug event* siap untuk melakukan *debugging event*. *Debugger* menggunakan *debugging event* berupa fungsi WaitForDebugEvent() dan fungsi ContinueDebugEvent(). WaitForDebugEvent() akan tetap menunggu hingga *event* muncul. Untuk setiap *event* yang ada pada *debugger* akan ditangani oleh *event handler* sebelum proses dilanjutkan. Disaat *handler* selesai dieksekusi, maka proses sebelumnya akan lanjut dieksekusi. Peran dari fungsi ContinueDebugEvent() muncul disaat proses selanjutnya dieksekusi, disamping itu peran lain dari fungsi ini ialah melanjutkan proses *exception*. *Exception* merupakan suatu indikasi munculnya keadaan *error* pada *debugger* ketika program dieksekusi. Beberapa bentuk *exception* yang ditangani oleh *debugger* yaitu berupa *exception* terhadap *breakpoint*, *exception* terhadap pelanggaran akses (*access violation*), serta *exception* terhadap izin akses yang tidak tepat pada memori. Sistem *debugger* kemudian melakukan *detach*/melepaskan proses apabila analisis proses selesai dilakukan.

Terdapat 4 kategori *exception* yang ditangani oleh Windows *debugger* yaitu:

1. *Exception access violation* merupakan *exception* yang muncul pada proses yang tidak diizinkan untuk mengakses memori. *Exception* ini ditanganimenggunakan sebuah *handler* yang dinamakan *access violation handler*.
2. *Exception breakpoint* merupakan *exception* yang muncul ketika *soft breakpoint*. *Exception* ini ditangani menggunakan *handler* yang dinamakan *handler breakpoint*.
3. *Exception guard page* merupakan *exception* yang terjadi karena mengakses *guard page* pada *memory breakpoint*. *Guard page* ialah sebuah izin memberi akses terhadap *page* pada memori. *Exception* ini ditangani menggunakan *handler.*
4. *Exception single step* merupakan *exception* yang muncul disaat menemukan *hardware breakpoint*. *Hardware breakpoint* menggunakan INT1 ketika menemukan *single-step* event. *Single-step* mengeksekusi instruksi secara satu satu. *Exception* ini ditangani menggunakan *handler* yang dinamakan *exception handler single step*.

## **Analisis Memori**

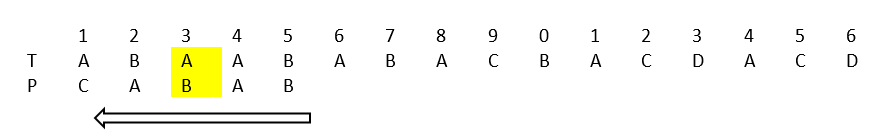
Pada penelitian ini, peneliti menganalisis *text section* dari perangkat lunak. Oleh karena itu, peneliti membangun sebuah *custom debugger* yang dirancang untuk mampu mengakses langsung *text section* program. Melalui informasi yang diperoleh dari *text section*, sistem dapat mendeteksi apakah perangkat lunak mengandung *malware* atau tidak. Microsoft menyediakan fasilitas berupa *library* *Portable Executable* (PE) *File* bagi pengembang yang ingin melakukan penelitian di ranah analisis memori. *Library* PE *File* ini memiliki fungsi sebagai PE *loader* untuk membuka *file executable* dan memuatnya kedalam memori serta mendapatkan DLL (*dynamic-link library*) dari kode program yang sudah dikompilasi. Ketika *file executable* telah dibuka, maka seluruh alamat instruksi dalam bentuk heksadesimal dari kode program dapat di peroleh. Fungsi PE *File* ini dilakukan pada level kernel. Pada Maret 1994, Matt Pietrek telah menjelaskan dalam artikel penelitiannya jika *text section* dapat diperoleh dengan menggunakan *library* PE *File* [35].

## **Algoritma *String Matching***

Didalam *shellcode* yang diduga mengandung *malware* terdapat suatu *string* yang sangat panjang. *String* tersebut akan digunakan peneliti untuk membandingkan *string* *text section* pada *file* *executable* dengan *string malware* dari *database* ClamAV yang telah ada. Terdapat 3 bentuk penerapan yang dirancang akan digunakan pada *string matching* pada *custom debugger* yaitu algoritma Boyer Moore, algoritma Rabin-Karp, dan struktur data *suffix tree*.

### **Algoritma Boyer-Moore**

Algoritma ini menerapkan dua aturan Boyer-Moore yaitu aturan *bad character* dan aturan *good shift*. Untuk lebih jelas, analisis algoritma akan diilustrasikan pada sebuah contoh pada deretan P terhadap T di Gambar 3.6.

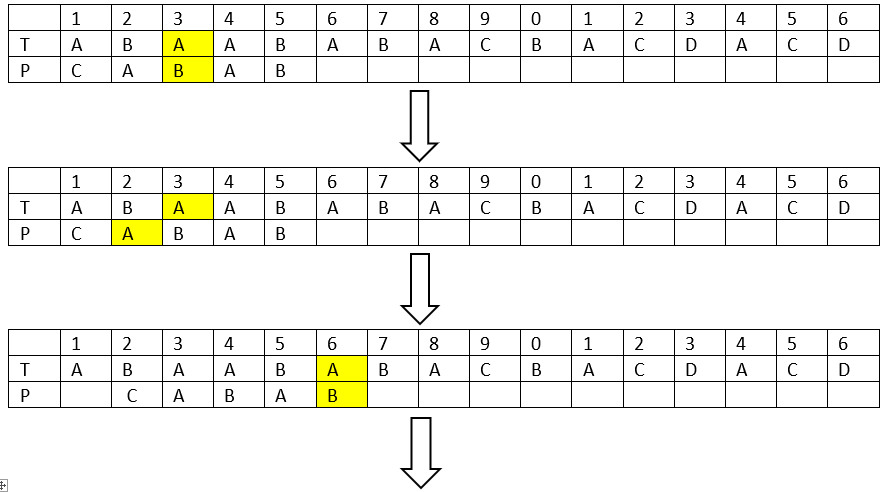


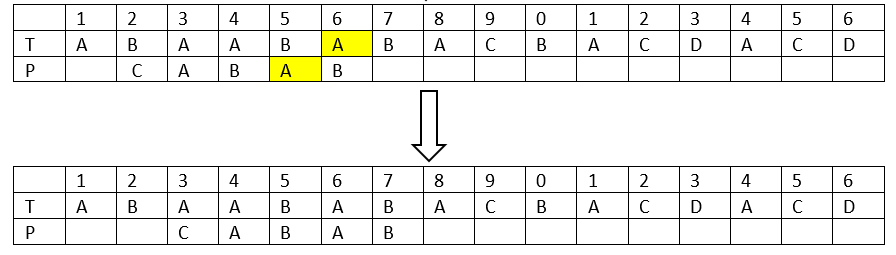
**Gambar 3.6 Contoh Algoritma Boyer-Moore**

Untuk memeriksa kemunculan P sebagai pola atau *string* yang dicari pada T sebagai teks atau *string* yang disediakan dengan bentuk posisi deretan diatas, algoritma Boyer-Moore akan mulai mengambil nilai dari ujung kanan P, kemudian bandingkan T(5) dengan P(5). Perolehan cocok, lalu bandingkan T(4) dengan P(4) dan selanjutnya, bergerak dari kanan ke kiri sampai menemukan ketidakcocokan pada T(3) dengan P(3). Ketika ketidakcocokan muncul maka deretan akan digeser ke suatu poin di T yang akan dijelaskan selanjutnya pada dibawah ini. Kemudian mulai lagi dengan membandingkan ujung kanan dari P.

Ada 2 pendekatan dari algoritma ini dalam menangani ketidakcocokan yaitu aturan *bad character* dan aturan *good shift*.

#### **Aturan Bad Character**

Contoh kasus aturan *bad character* dapat dilihat pada ***Gambar 3.7*** di halaman berikut sebagai bahan analisis yaitu ketika ketidakcocokan terjadi diantara karakter P dan karakter T. Contoh Gambar 3.7 merupakan penerapan pendekatan ini dilakukan dengan cara menggeser P lebih dari satu karakter. 



**Gambar 3.7 Contoh Bad Character**

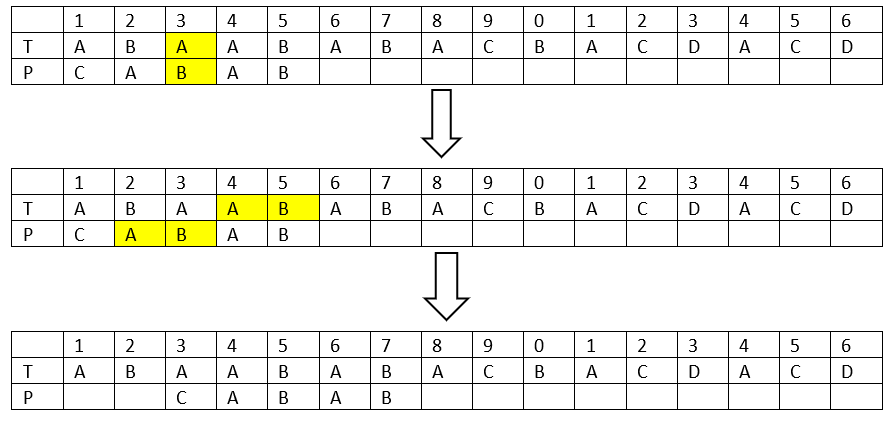
Pada gambar diatas dapat dilihat P adalah *pattern* yang akan dicari pada teks dan T adalah *Text*. Tahap pertama, dicek kecocokan karakter antar T dan P. Dicek dari P(5) dan T(5) dan hasilnya cocok, dilanjut lagi ke P(4) dan T(4) dan hasilnya cocok. Kemudian pada P(3) dan T(3) tidak cocok dan dicari kecocokan pada P yang karakternya sama dengan karakter T(3). Dari hasil pencarian, didapati karakter pada T(3) sama dengan karakter P(2). Maka letak pola akan berubah posisi dimana akan ditandai karakter P(2) akan diletakkan tepat dibawah posisi T(3).

Selanjutnya pada tahap kedua, lakukan pengecekan kembali dimulai dari karakter terakhir pola yang akan dicari. Jika dari ilustrasi gambar diatas, pencocokan dimulai dari indeks T(6) dan P(6) dan hasilnya tidak cocok. Kemudian dicari pada P, karakter yang sama dengan T(6) dan ditemukan pada P(5). Kemudian P(5) diletakkan tepat dibawah T(6). Kemudian dilakukan pengecekan pada P(7) dan T(7) dan hasilnya cocok. Demikianlah proses selanjutnya, sama dengan proses sebelumnya sehingga didapatkan hasil bahwa *pattern* berada pada *text* atau tidak.

#### **Aturan Good Shift**

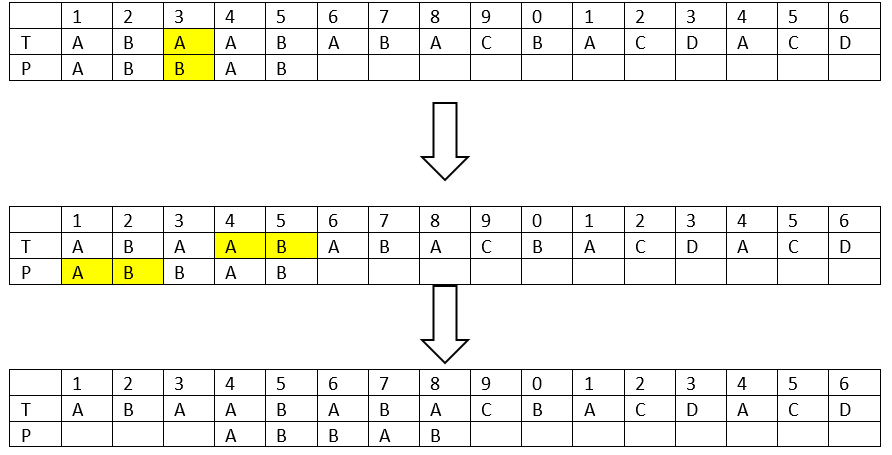
Sementara untuk contoh kasus aturan *good shift* diilustrasikan dibawah ini.

Kasus 1: kemunculan dari t pada P cocok dengan t pada T. Dimana pola P mengandung kemunculan t lain. Sehingga, akan dicoba penggeseran P ke deretan dimana munculnya t pada T.

Pada kasus 1, *Substring* t pada T(4) dan T(5) cocok dengan P(4) dan P(5) sebelum ketidakcocokan pada indeks 3. Selanjutnya, cari kemunculan lain dari t pada P. Kita menemukan kemunculan pada posisi P(2) dan P(3) maka geser ke kanan pola tersebut sehingga sederetan dengan T(4) dan T(5).

**Gambar 3.8 Kasus 1 Good Shift**

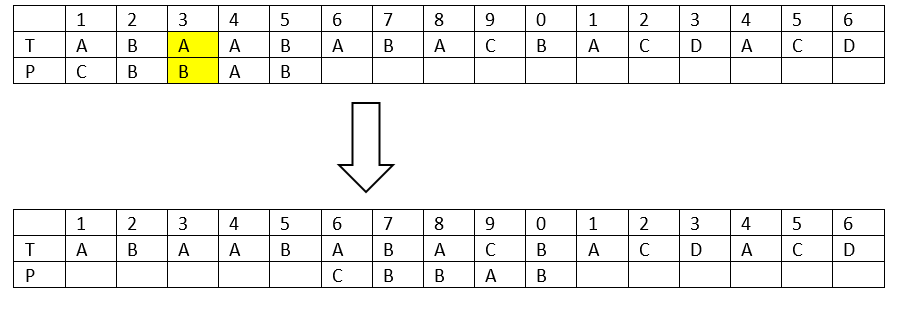
Kasus 2: prefiks dari P yang cocok dengan *suffix* t pada T. Ada suatu pengecualian kondisi dimana *substring* t tidak muncul pada P sehingga yang dilakukan adalah melakukan pencarian terhadap *suffix* atau substring t lain yang bisa dicocokkan dengan *substring* t pada T.



**Gambar 3.9 Kasus 2 Good Shift**

Pada kasus 2, *substring* t (“AB”) cocok dengan P pada indeks 4-5 sebelum ketidakmunculan terjadi. Namun karena t tidak ada kemunculan lagi pada P maka cari prefiks lain dari P yang cocok dengan *suffix* dari t pada T. *Substring* yang mungkin dibandingkan ialah t “AB” (*highlight* kuning) yang dimulai dari indeks 1. Sehingga P akan digeser sehingga sederetan prefiks dengan *suffix*.

Kasus 3: P digeser melewati t. Jika kondisi kasus 2 tidak terpenuhi, maka geser P hingga melewati t. Pada kasus 3, tidak ada kemunculan dari t (“AB”) pada P dan tidak ada prefiks P yang cocok dengan *suffix* t. Sehingga, kasus tersebut diatasi dengan menggeser P melewati t pada T dimana letaknya diawali pada indeks ke 6. Dapat dilihat gambarnya pada Gambar 3.10.



**Gambar 3.10 Kasus 3 Good Shift**

Algoritma Boyer-Moore ini efisien digunakan untuk pencarian *string* karena algoritma ini bekerja sangat cepat ketika *string* yang diberi panjang dan pola yang dicari juga panjang. Kecepatan algoritma ini berasal dari teknik pergeseran pola yang sangat banyak dari kanan ke kiri. Aturan *bad character shift* merupakan teknik pergeseran yang menghindari perulangan ketidakcocokan karakter terhadap karakter target/teks. Sementara aturan *good suffix shift* merupakan teknik untuk menemukan karakter pola yang cocok terhadap target karakter yang sudah cocok. Pada sistem *string matching* yang peneliti terapkan adalah aturan *bad character shift* karena seperti diketahui bahwa *signature* berisi kumpulan karakter yang tidak beraturan baik huruf maupun angka sehingga yang tipe data-nya dibuat sebagai *string* sehingga sulit menggunakan aturan *good suffix shift* pada karakter yang beraturan tersebut.

### **Algoritma Rabin-Karp**

Algoritma ini membandingkan nilai *hash* dari *pattern* (*substring* yang akan dicari) dan nilai *hash* dari teks yang akan diperiksa. fungsi hash ini digunakan untuk mengkonversi *string* menjadi bilangan yang disebut *hash value.* Pada tahap pertama dilakukan pemenggalan karakter *string* sebanyak panjang pola *string* yang dicari, kemudian kalkulasi nilai hash *substring* yang dicari dan nilai *hash* dari string yang disediakan *file* sampel. Bandingkan kedua hasil perolehan *hash*. Jika sama, bandingkan *substring* dan pola yang dicari. Jika tidak sama, geser satu karakter ke karakter selanjutnya dan ambil panjang *string* yang sama dengan panjang pola yang dicari. Kembali lakukan kalkulasi nilai *hash* antar keduanya. Lakukan iterasi hingga diperoleh *string signature* dan *string* *library* cocok.

Contoh paling sederhana adalah dengan menjumlahkan bilangan ASCII nya. Misalnya, terdapat *string* “Strategi”, dan kita ingin mencari kata “rat” pada *string* tersebut.

1. Langkah pertama adalah menentukan panjang *string* sumber dan *string* yang ingin dicari. Dalam kasus ini, panjang *string* sumber n = 8. Sedangkan *string* yang ingin dicari panjangnya m = 3.
2. Langkah kedua adalah dengan mengubah kata yang ingin dicari dengan menggunakaan fungsi *hash*. Jadi nilai *hash* dari “rat” adalah 327 (nilai ASCII r =114, a=97, dan t=116).
3. Kemudian melakukan iterasi dari indeks i = 0 sampai i = n-m+1. Pada saat iterasi, dilakukan perbandingan nilai *hash* dari nilai *hash* kata yang ingi dicari dengan nilai *hash* dari *string* sumber pada indeks ke i sampai i+m-1. Bila sama, maka akan mengembalikan nilai *true*, tapi jika tidak, maka perbandingan akan dilakukan dengan indeks berikutnya. Dalam hal ini, kita membandingkan 327 dengan nilai *hash* dari *string* “Str” (nilai *hash* = 313). Karena tidak sama, maka dilakukan perbandingan dengan nilai *hash* dari indeks selanjutnya. Pada kasus ini indeks ke dua sampai ke 4(i+m-1). Pencocokan akan dilakukan terus sampai mendapatkan nilai *hash* yang sama seterusnya.

### **Struktur Data *Suffix Tree***

Proses *substring matching* dilakukan dengan menerapkan algoritma *suffix tree* dimana signature *malware* dari *library* ClamAV akan direpresentasikan sebagai *suffix*. Proses *substring matching* dari algoritma ini terdiri dari beberapa langkah yaitu: proses pemenggalan *string* untuk mendapatkan *suffix*, pembangunan *suffix tree*, serta proses *substring matching* dari *suffix tree* yang sudah dibangun sebelumnya.

#### **Proses Pemenggalan String**

Proses menemukan *suffix* ini didahului dengan pembacaan tiap *signature malware* dari *file* sampel. Sehingga input dari algoritma ini adalah sebuah *string*, kemudian dilakukan pemeriksaan semua kemungkinan bentuk kata dengan cara melakukan pemenggalan *string* yang membuat *suffix* baru dari *string* yang dianalisis. Proses pemenggalan *string* dilakukan dengan cara memotong karakter dari bagian *string* tersebut, karakter disini akan disebut sebagai *substring*. Illustrasi dari proses ini akan dijelaskan pada suatu contoh *signature malware*. *Suffix tree* dari *string* S = CAGTCAGG$. *String* S memiliki tujuh *suffix*, yaitu CAGTCAGG$, AGTCAGG$, GTCAGG$, TCAGG$, CAGG$, AGG$, GG$ dan G$ dimana akan dinomori dari 1 hingga 8. Melalui *suffix tree*, kita dapat memeriksa apakah *suffix string* tersebut merupakan *substring* dari S.

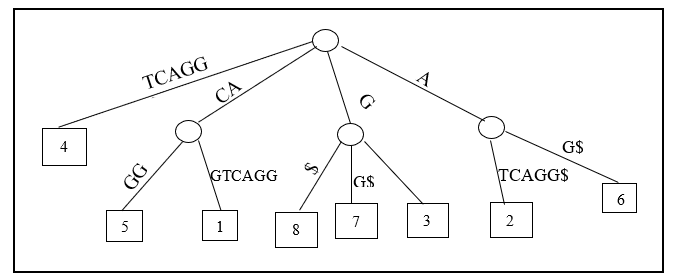
Penggal *string* per satu karakter prefiks untuk mendapatkan *substring*. Hasil *substring* sebanyak n karakter dari *string*. Sebagai contoh untuk *string* = CAGTCAGG$. Pemenggalan *string* tersebut dapat dilihat pada Tabel 3.2.

# **Tabel 3.2 Pemenggalan *String***

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Nomor | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Suffix | CAGTCAGG$ | AGTCAGG$ | GTCAGG$ | TCAGG$ | CAGG$ | AGG$ | GG$ | G$ |

#### **Proses Pembangunan Suffix Tree**

Suffix *tree* dari *string* S adalah pohon yang sisi-sisinya diberi label *string* yang merupakan sufiks dari S dan sesuai dengan tepat satu jalur dari akar ke daun. Berikut contoh proses pembangunan *suffix tree* jika diberikan *signature malware* yaitu *text section* dari *file executable* sebagai pola yang akan dicari pada *signature* yang sudah disediakan sebelumnya. *Signature malware* dari *file* sample sebagai *string*. Jika diberikan *string* S = CAGTCAGG$ dan pola yang akan dicari adalah CAG. Kontruksi *suffix tree* dimulai dengan membentuk akar dan *leaf* bernomor 1, dihubungkan dengan *edge* bernama S$. Kemudian masukkan *suffix* ke dalam *tree* dengan ketentuan sebagai berikut: Periksa apakah sudah ada suatu *node* dengan nilai *tree* yang sama dengan karakter yang ingin dimasukkan. Caranya ikuti jalur dari akar ke karakter yang memiliki karakter serupa dengan *suffix* yang akan dimasukkan. Jika *edge* belum merepresentasikan *suffix* tersebut, maka terjadi *split* atau penambahan *node*. Dengan demikian, melalui ilustrasi penerapan *suffix tree* sebelumnya hasil pencarian relatif cepat karena pencarian *substring* dilakukan pada *tree*. Pencarian dengan teknik ini membutuhkan banyak *resource memory* untuk membangun *tree*. Namun pencarian ini mampu langsung mencari karakter pertama *suffix* pada *tree*. Melalui *suffix* yang diperoleh sebelumnya akan digunakan pada pembentukan suffix tree seperti Gambar 3.11.



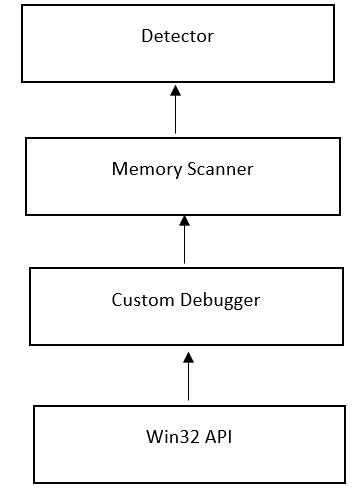
**Gambar 3.11 Suffix Tree**

#### **Proses String Matching**

Setelah melakukan konstruksi *suffix tree* untuk S. Peneliti akan mengambil *text section file executable* sebagai pola yang digunakan pada pencarian *string* dari *suffix tree*. Pola tersebut yang merupakan sebuah *string* lalu akan dicocokkan dengan *path* ke *edge* pada *tree* dimulai dari *root*-nya. Terdapat tiga kondisi yang terjadi pada saat substring matching. Kondisi pertama, apabila pola tidak cocok artinya pola tersebut tidak ada pada *tree*. Kondisi kedua, apabila kecocokannya cocok pada *node* tertentu maka pola tersebut ada dan semua *leaf* yang berbeda *edge* merepresentasikan pola yang dicari. Kemudian kondisi terakhir, apabila kecocokannya cocok pada suatu *edge* maka pola tersebut ada *edge* yang telah ditemukan itu.

# **RANCANGAN**

Pada bab ini dijelaskan proses perancangan yang dilakukan untuk pembangunan *debugger* sebagai pendeteksi *malware* menggunakan algoritma *memory scanning*. Adapun arsitektur *debugger* yang akan dibangun dapat dilihat pada Gambar 4.1.



**Gambar 4.1 Arsitektur Sistem**

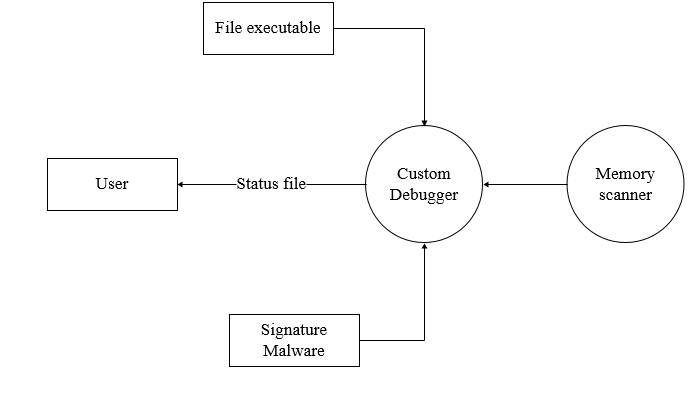
Arsitektur sistem yang akan dibangun dalam pengerjaan Tugas Akhir ini adalah *custom debugger* yang mengimplementasikan algoritma *memory scanning*. Oleh karena itu, sistem membutuhkan alokasi *resource* memori dari *processor* dan memori Win32 API komputer yang digunakan pada proses *memory scanning*. *Custom debugger* ini akan dijalankan diatas *virtual box.* Pada *custom debugger* ditambahkan algoritma yang digunakan untuk pengambilan memori (*snapshot memory*).Selanjutnya algoritma *memory scanning* akan mendapatkan *text section* dari *file executable* yang diduga mengandung *malware*. *Detector* digunakan untuk memeriksa *file executable* merupakan *malware* atau tidak dapat disimpulkan *malware*. *Detector* pada sistem ini akan mengimplementasikan algoritma *string matching* yang mampu membandingkan *signature malware* yang berada di *text section* dengan *signature* *malware* pada *library* berisi *malware*. Jika *signature* sama maka *file executable* tersebut mengandung *malware*. Sebaliknya, jika tidak sama maka *file executable* tidak Dapat disimpulkan mengandung *malware* atau tidak.

## **DFD Rancangan Sistem**

Pada bagian ini dijelaskan bagaimana gambaran dari aliran data pada sistem yang akan dibangun. Aliran data tersebut digambarkan secara granular dalam bentuk *data flow diagram* (DFD).

### **DFD Level 0**

Berikut DFD level 0 untuk sistem yang akan dibangun dapat dilihat pada Gambar 4.2.

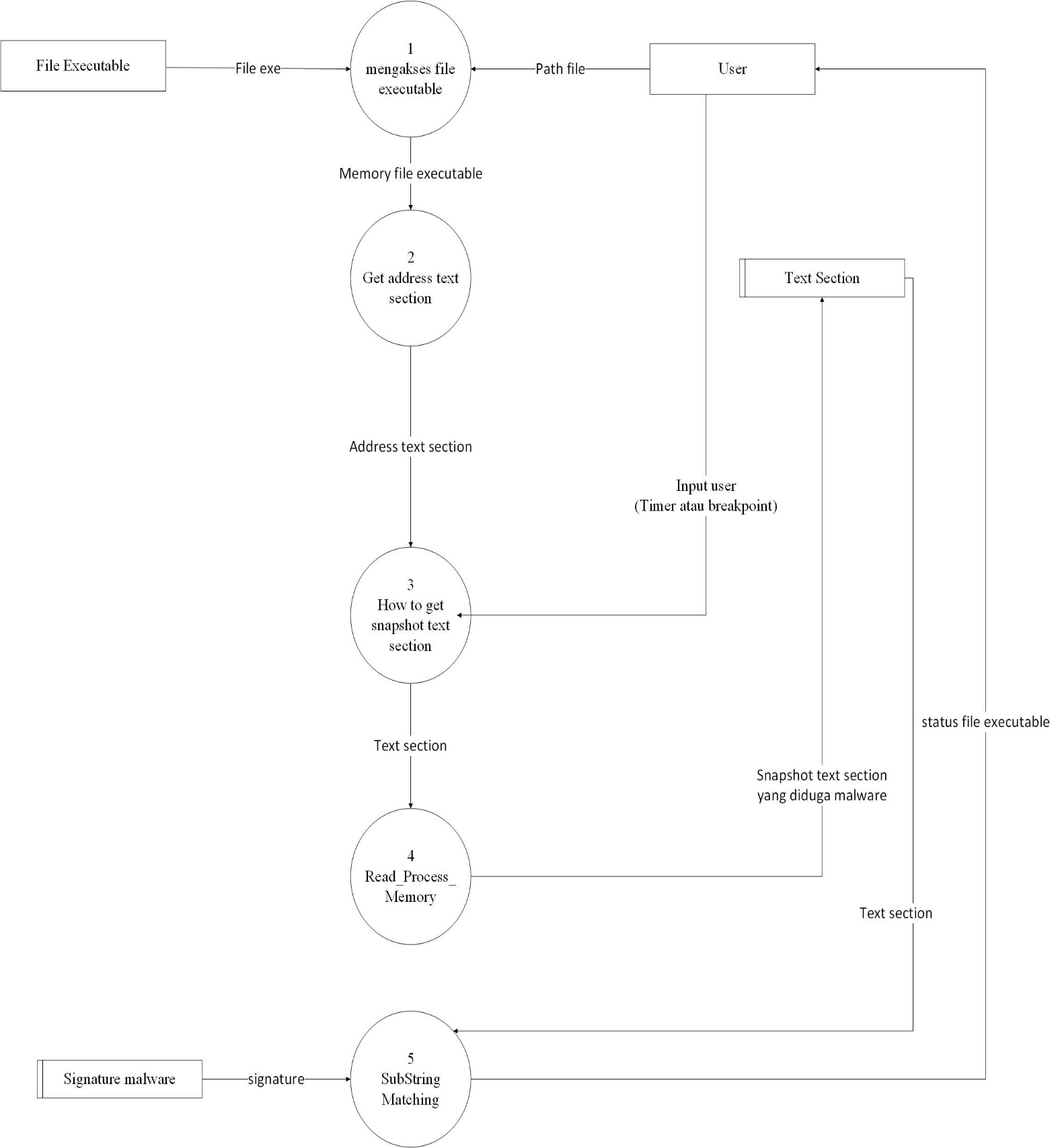


# **Gambar 4.2 DFD Level 0**

*Debugger* yang dibangun akan mengakses *text section* dari *file executable* dan dan *signature malware* yang ada. Pada *custom debugger* tersebut telah terdapat me*mory scanner* yang digunakan untuk membandingkan *text section* dan *signature malware* kemudian hasilnya berupa status, apakah *file executable* mengandung *malware* atau tidak dapat disimpulkan.

### **DFD Level 1**

*Data Flow Diagram* level 1 untuk sistem yang akan dibangun dapat dilihat pada Gambar 4.3 dihalaman berikutnya.

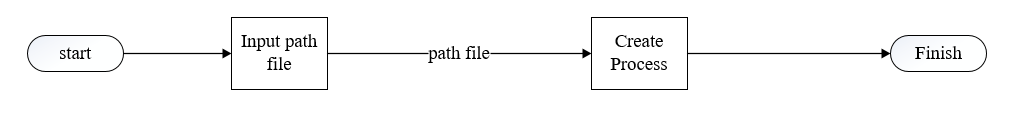


# **Gambar 4.3 DFD Level 1**

Setiap proses pada DFD Level 1 akan dibahas secara rinci pada subbab dibawah ini.

#### **4.1.2.1 Mengakses File Executable**

Berikut adalah proses mengakses *file executable* yang akan diperiksa apakah mengandung *malware* atau tidak. Cara yang digunakan dalam mengakses *file executable* dengan *load process*. *Load process* digunakan untuk membuat proses pada *file executable* dengan menginput *path* *file executable* tersebut. Setelah di *input path file* maka pada proses ini dapat mengakses *file executable* dan dapat mengakses isi memori dari *file* tersebut dengan menggunakan fungsi *CreateProcess*(). Untuk *flowchart* pada proses ini dapat dilihat pada Gambar 4.4.



**Gambar 4.4 Flowchart Mengakses File Executable**

Untuk *pseudocode* pada *create process* ini dapat dilihat pada *Pseudocode* 4.1.

**Pseudocode 4.1 Create Process**

Function load(self, path\_to\_exe):

Inisialisasi creation\_flags 🡨 DEBUG\_PROCESS

Inisialisasi startupinfo 🡨 STARTUPINFO()

Inisialisasi process\_information 🡨 PROCESS\_INFORMATION()

Inisialisasi startupinfo.dwFlags 🡨 0x1

Inisialisasi startupinfo.wShowWindow 🡨 0x1

Inisialisasi startupinfo.cb 🡨 sizeof(startupinfo)

If(kernel32.CreateProcessA(path\_to\_exe,None,None, None, None, creation\_flags, None, None, byref(startupinfo),byref(process\_information))):

Print “[\*] We have successfully launched the process ! “

Print “[\*] The Process ID I have is: %d “ % process\_information.dwProcessId

self.pid 🡨 process\_information.dwProcessId

self.h\_process 🡨 self.open\_process(process\_information.dwProcessId)

self.debugger\_active 🡨 TRUE

Else:

Print “[\*]Error with error code %d“ % kernel32.GetLastError()

#### **4.1.2.2 Get Address Text Section**

Pada proses ini, menggunakan fungsi dari *library* PE *File* untuk mendapatkan alamat dari *text section*. *Input* pada proses ini adalah memori dari *file* dan akan dicari *address* dari *text section*. Dalam fungsi PE *File*, akan ditentukan *size* dari *text section* (size\_of\_section), *base relocation*, dan alamat virtual *text section*. *Size of section* digunakan untuk menentukan *size* dari alamat *text section*, *base relocation* merupakan alamat awal pada *text section* setelah dialokasikan dan alamat virtual dari *text section*. *Output* pada fungsi PeFile() ini adalah alamat *text section* dan *size* dari *text* *section*. Untuk *pseudocode* pada proses mendapatkan alamat *text* *section* dapat dilihat pada *Pseudocode* 4.2.

**Pseudocode 4.2 Get Address Text Section**

Function getAddress(path):

Inisialisasi pe 🡨 objek pefile.PE(path)

Inisialisasi size\_of\_section 🡨 pe.sections[0].Misc\_VirtualSize

Inisialisasi base\_relocation 🡨 pe.OPTIONAL\_HEADER.ImageBase

Inisialisasi temp 🡨 array dengan isi pe.sections[0].VirtualAddress + base\_relocation dan size\_of\_relocation

Return temp

#### **4.1.2.3 Mendapatkan Snapshot Text Section**

Pada proses ini, pertama sekali di *set breakpoint* pada awal dari *address text section* sebagai penanda *address of entry point* dari *text section* tersebut. Pada *set breakpoint* ini, menggunakan konsep *soft breakpoint*, dimana instruksi dari alamat awal *text* *section* diganti dengan INT3 atau 0xCC. Instruksi alamat yang sudah diganti tersebut akan disimpan pada *breakpoint dictionary* sebagai penyimpanan instruksi sementara. Instruksi akan dikembalikan ke alamat sebelumnya ketika terjadi kondisi *exception*. Setelah menentukan *breakpoint* pada awal dari *address text section*, selanjutnya ke proses untuk mengambil *snapshot* dari *text section*. Ada 2 cara yang dilakukan untuk mendapatkan *snapshot* dari *text section*, satu dengan menggunakan *timer* dan satu lagi dengan meletakkan *breakpoint* pada sembarang *address* secara random. Untuk *pseudocode* men-*set breakpoint* dapat dilihat pada *Pseudocode* 4.3.

**Pseudocode 4.3 Menentukan Breakpoint**

Function bp\_set (self,address):

Print “[\*] Setting breakpoint at: 0x%08x” % address

If not breakpoints menggunakan address dari variabel has\_key:

Inisialisasi variabel old\_protect 🡨 tipe data c\_ulong bernilai 0

Kernel32.VirtualProtectEx(self.h\_process, address, 1, PAGE\_EXECUTE\_READWRITE, byref(old\_protect))

Inisialisasi variabel original\_byte 🡨 byte dalam address

If original\_byte != False:

If fungsi write\_process\_memory menggunakan address dan “\xCC”:

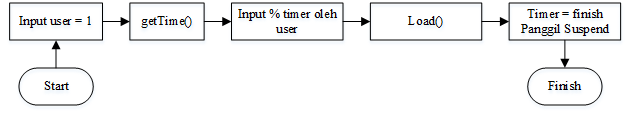
Breakpoints pada index address 🡨 dengan original\_byte

RETURN True

Else:

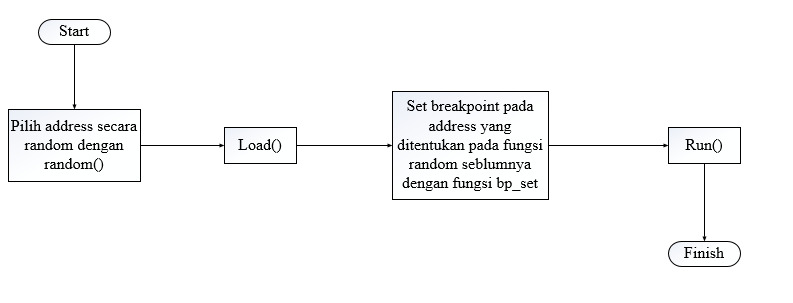
Return False

*User* akan diminta untuk memilih dengan kedua cara tersebut. Cara pertama dengan *timer*. Pertama sekali, menggunakan fungsi getTime() yang digunakan untuk menghitung waktu yang digunakan untuk memproses seluruh *text section*. Kemudian *user* akan diminta untuk menginput berapa persen dari total waktu yang ada. Dimana dalam rentan waktu yang diinput *user* tersebut, pada waktu tersebut *snapshot* *text section* akan diambil untuk diproses selanjutnya. Pada proses tersebut, menggunakan fungsi load() untuk *create* *process* pada saat *timer* dijalankan. Jika *timer* yang ditentukan telah selesai, maka akan dipanggil fungsi suspend() untuk menghentikan proses sehingga *snapshot* dapat diambil dan akan digunakan pada proses selanjutnya. Kemudian *debugger* di *run*. Untuk *flowchart* pada proses *timer* ini dapat dilihat pada Gambar 4.5.



**Gambar 4.5 *Flowchart Timer***

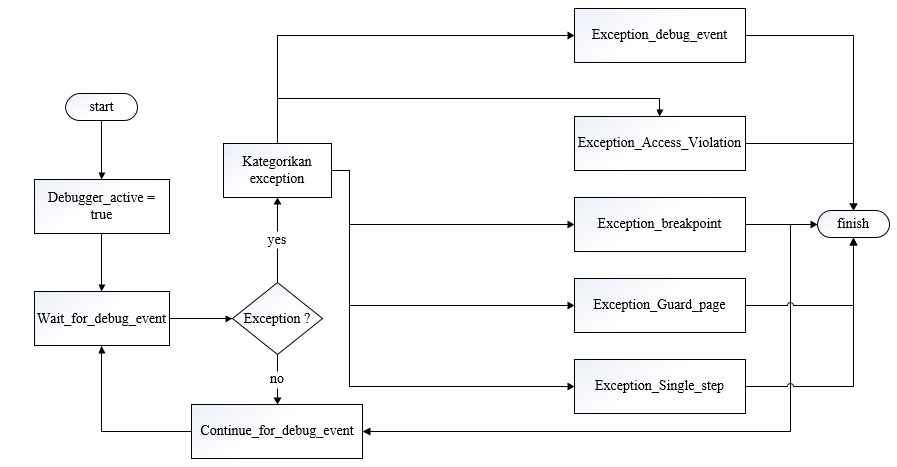
Kemudian cara kedua yang digunakan untuk mengambil *snapshot text section*, yaitu dengan cara men-set *breakpoint* pada sembarang *address* atau secara acak. Pada proses ini menggunakan fungsi random() dengan parameter *size* dari *text section* yang didapat dari proses sebelumnya. Fungsi random() ini digunakan untuk mengambil sembarang *address text section* kemudian menggunakan fungsi load() untuk melakukan *create process*. *Breakpoint* yang ditentukan secara random, merupakan penanda bahwa debugger akan diberhentikan dan akan diambil *snapshot* dari awal *text section* sampai *address text section* yang ditentukan. Selanjutnya dipanggil fungsi bp\_set() untuk men-*set breakpoint* pada *address* yang telah dipilih secara acak pada proses sebelumnya. Pada fungsi bp\_set ada fungsi read\_process\_memory yang digunakan untuk membaca isi *text section* dan menulisnya kemudian menggunakan fungsi *run* untuk menjalankan *debugger*. Untuk *flowchart* pada proses ini dapat dilihat pada Gambar 4.6.



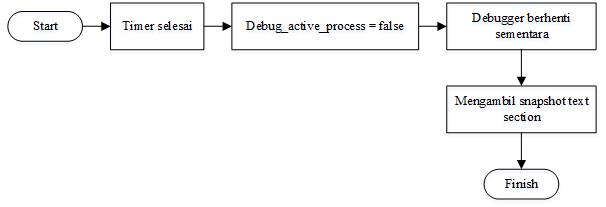
# **Gambar 4.6 *Flowchart* Penentuan *Breakpoint* secara Acak**

##### **4.1.2.4.5 *Execute Code***

Pada proses ini menggunakan fungsi run() yang dipanggil pada saat metode timer dan set breakpoint secara random. Proses ini menggunakan variabel debugger\_active dan get\_debug\_event. Jika variabel debugger\_active = *true* maka akan dipanggil variabel get\_debug\_event. Get\_debug\_event merupakan fungsi untuk mengambil debug\_event sehingga kode dapat dijalankan. Ada 2 *event* dalam get\_debug\_event yaitu wait\_for\_debug\_event dan continue\_for\_debug\_event. Setiap instuksi yang dieksekusi akan diperiksa apakah mengandung *exception* atau tidak. Pada saat yang bersamaan, *event* yang terjadi adalah wait\_for\_debug\_event. Jika tidak mengandung *exception* maka akan dijalankan continue\_for\_debug\_event dan memeriksa instruksi selanjutnya. Untuk *flowchart* pada proses ini dapat dilihat pada Gambar 4.7 di halaman selanjutnya.

**Gambar 4.7 *Flowchart Execute Code***

Melalui ilustrasi dijelaskan apabika ditemukan *exception* maka akan ada *handler* untuk mengatasi setiap *exception.* Setelah proses *timer* dan *set breakpoint* selesai maka akan digunakan fungsi suspend() yang digunakan untuk menghentikan *debugger* sementara sehingga isi dari memori dapat digunakan pada proses *string matching*. Pada fungsi suspend ini, menset debug\_active\_process menjadi *false*. Dengan menset fungsi ini menjadi *false*, sehingga proses dapat berhenti untuk sementara sehingga proses pengambilan *snapshot memory* dapat dilakukan. Untuk *flowchart* dari fungsi ini dapat dilihat pada Gambar 4.8 .



**Gambar 4.8 Flowchart Suspend debugger**

Untuk *pseudocode* pada proses *suspend* dapat dilihat pada *Pseudocode* 4.4.

**Pseudocode 4.4 Suspend Process**

Function suspend(self):

self.debugger 🡨 False

self.shellcode 🡨 self.read\_process\_memory(self.address\_text\_section, self.size\_text\_section)

#### **4.1.2.4 Read Process Memory**

Fungsi read\_process\_memory() digunakan untuk membaca *text section* setelah dilakukan proses *suspend*. Untuk pseudocode pada proses ini dapat dilihat pada Pseudocode 4.5.

Function read\_process\_memory(self,address,length):

data 🡨 “”

read\_buf 🡨 create\_string\_buffer(length)

count 🡨 c\_ulong(0)

kernel32.ReadProcessMemory(self.h\_process, address, read\_buf, length, byref(count))

data 🡨 read\_buf.raw

RETURN data

**Pseudocode 4.5 Read Process Memory**

#### **4.1.2.5 Deteksi Signature dengan Algoritma String Matching**

Pada proses *string matching* ini, *signature* pada *library malware* dibandingkan dengan *snapshot* *text section* yang telah didapat pada proses sebelumnya. Ada beberapa algoritma yang dapat dipilih dalam *substring matching*, seperti algoritma Boyer-Moore, algoritma Rabin-Karp, dan Suffix Tree.

##### **4.1.2.5.1 Algoritma Boyer-Moore**

Pada algoritma ini, peneliti menggunakan pendekatan *bad character heuristic*. Langkah-langkah yang dilakukan pada pendekatan ini, yang pertama adalah membandingkan ketidakcocokan antara *pattern* dan *shellcode text section*. Perbandingan ini per karakter antara *pattern* dan *shellcode text section*. Perbandingan ini dilakukan dari kanan ke kiri. Setelah dibandingkan, maka *pattern* digesar melewati karakter yang tidak cocok. Kemudian di cek lagi, apakah *pattern* telah sama dengan *shellcode text section*.

Proses pengecekan dan pergeseran tiap karakter akan dilakukan sampai ditemukan kecocokkan antara *shellcode text section* dan *signature malware*.

*Pseudocode* dari algoritma Boyer-Moore dapat dilihat pada *Pseudocode* 4.6.

FUNCTION badCharHeuristic(string, size):

NO\_OF\_CHARS **🡨** 10000

badChar **🡨** [-1]\*NO\_OF\_CHARS “alokasi array sebanyak NO\_OF\_CHARS”

FOR i IN range(size):

badChar[ord(string[i])] **🡨** i

return badChar

FUNCTION boyer\_moore(txt, pat):

m **🡨** len(pat) “Panjang string pattern”

n **🡨** len(txt) “Panjang string text”

badChar **🡨** badCharHeuristic(pat, m)

s **🡨** 0

WHILE (s <= n - m):

j **🡨** m – 1

WHILE (j >= 0 and pat[j] == txt[s + j]):

j **🡨 j -** 1

IF (j < 0):

Return 1 “return 1 jika pattern terdapat pada text”

ELSE:

s **🡨 s +** max(1,j-badChar[ord(txt[s+j])])

Return 0

**Pseudocode 4.6 Algoritma Boyer-Moore**

##### **4.1.2.5.2 Algoritma Rabin-Karp**

Pada algoritma ini menggunakan fungsi hash() yang digunakan untuk mengkonversi *string* menjadi bilangan yang disebut *hash value*. Contoh hash value dari string “hello” adalah 5. Langkah-langkah yang digunakan dalam algoritma ini adalah yang pertama menentukan panjang *string* *signature* *malware* dan *shellcode text section* yang ingin dicari. Kemudian menggunakan fungsi hash() untuk mengubah *shellcode text section*. Kemudian membandingkan nilai *hash* dari *shellcode text section* dan string *signature malware*. Jika hasilnya sama maka dapat dipastikan *file executable* mengandung *malware*, sebaliknya jika tidak sama maka tidak dapat dipastikan *file* tersebut mengandung *malware* atau tidak. Untuk *pseudocode* pada algoritma Rabin-Karp dapat dilihat pada *Pseudocode* 4.7 dihalaman selanjutnya.

**Pseudocode 4.7 Algoritma Rabin Karp**

FUNCTION rabin\_karp(pat, txt, q):

M **🡨** len(pat) “Panjang string pattern”

N **🡨** len(txt) “Panjang string text”

i **🡨** 0 “Counter”

j **🡨** 0 “Counter”

p **🡨** 0 “variabel penampung nilai hash pattern”

t **🡨** 0 “variabel penampung nilai hash text”

h **🡨** 1

FOR i IN xrange(M):

p **🡨** (d\*p + ord(pat[i])) % q “Menghitung nilai hash pattern”

t **🡨** (d\*t + ord(txt[i])) % q “Menghitung nilai hash text”

FOR i in xrange(N-M+1):

IF (p==t):

FOR j IN xrange(M):

IF (txt[i+j] != pat[j]):

Break

j **🡨 j +** 1

IF(j==M):

Return 1 “return 1 jika pattern terdapat pada text”

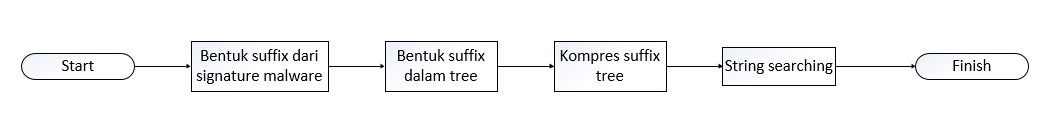
IF(i<N-M):

t **🡨** (d\*(t – ord(txt[i])\*h) + ord(txt[i+M]))%q

IF (t<0):

t **🡨** t + q

##### **4.1.2.5.3 *Suffix Tree***

Pada algoritma ini, *signature* dalam *library* dibuat dalam bentuk *suffix tree* dan *shellcode* *text section* akan dibanding ke *tree* tersebut. Langkah-langkah pada pembuatan *suffix tree* adalah, pertama sekali men-*generate* semua *suffix* dari *text* atau *signature malware*. Kemudian pertimbangkan semua *suffix* menjadi kata – kata individual dan buat sebuat *tree* yang telah dikompres. Kemudian untuk metode *string searching* yang dimulai dari root tree kemudian ke setiap *edge* dari *tree* tersebut. Untuk *flowchart* pada algoritma ini dapat dilihat pada Gambar 4.9*.*

**Gambar 4.9 Flowchart Suffix Tree**

# **IMPLEMENTASI**

Pada bab ini diuraikan hasil implementasi berupa batasan implementasi, implementasi *debugger* dengan algoritma *memory scanning* yang telah dianalisis dan dirancang sebelumnya.

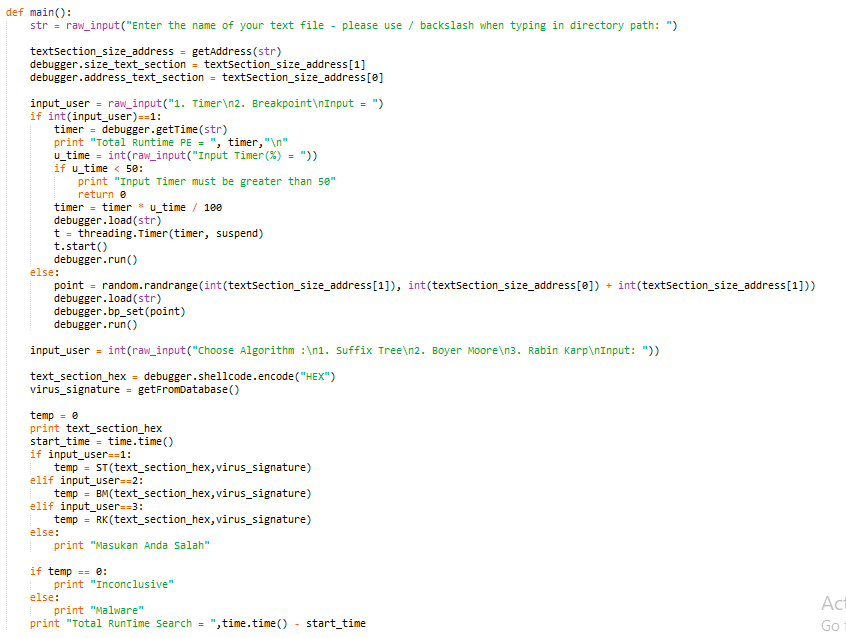
## **Batasan Implementasi**

Hal-hal yang dijadikan batasan dalam melakukan implementasi *In Memory Process Scaning for* *Malware Detection* adalah sebagai berikut:

1. *File executable* yang digunakan ada 2 jenis *file*, yaitu *file* yang mengandung *malware* dan yang tidak mengandung *malware*. Dimana *file* yang mengandung *malware* tersebut ada yang tidak terdeteksi oleh antivirus biasa tetapi dapat dideteksi oleh *debugger* yang dibangun peneliti.
2. *Signature malware database* yang digunakan berasal dari ClamAV.
3. Bahasa pemrograman yang digunakan untuk mengimplementasikan *debugger* ini yaitu bahasa Python.

## **Implementasi *Debugger* dengan Algoritma *Memory Scanning***

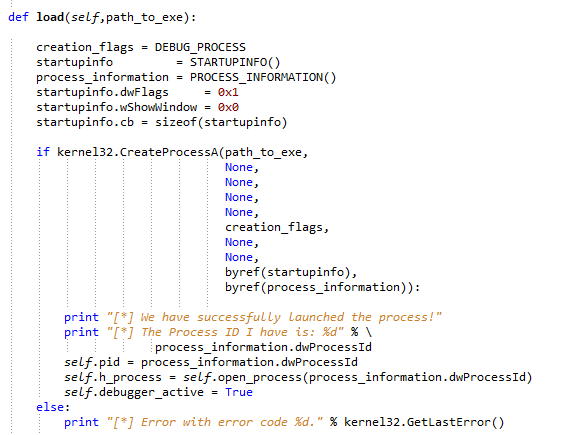
Pada bab ini akan dijelaskan tentang pembangunan *debugger* dengan algoritma *memory scanning*. Adapun proses yang terjadi adalah mengakses *file executable*, mengakses *text section*, mengambil *snapshot text section* dan *substring matching*. Untuk kode program dapat dilihat pada Gambar 5.1 yang didefinisikan dengan fungsi main()pada halaman berikutnya.



**Gambar 5.1 Implementasi main()**

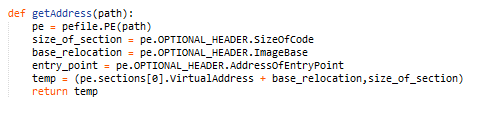
### **Implementasi untuk Mengakses *File Executable***

# Pada implementasi ini, cara untuk mengakses *file executable* dengan *load process*. Fungsi ini digunakan untuk membuat proses dan menginisialisasi setiap atribut yang digunakan dalam *load process*, fungsi ini memanggil *library* kernel32 *create process*. Pada fungsi ini, membutuhkan *path* dari *file* tersebut sehingga *file*-nya dapat diakses. Kode program fungsi ini dapat dilihat pada Gambar 5.2.

 **Gambar 5.2 Implementasi *Load Process***

### **Implementasi untuk Mengakses *Text Section***

Pada implementasi ini, akan dicari alamat dari *text section* dan *size* dari alamat sehingga *debugger* hanya akan memeriksa *text section* dari memori *file* tersebut. Pada proses ini menggunakan *library* PE *File* untuk *file executable*. Keluaran dari fungsi ini berupa *array* yang berisi alamat *text section* dan *size* *text section*. Kode program fungsi ini dapat dilihat pada Gambar 5.3 dihalaman berikutnya.

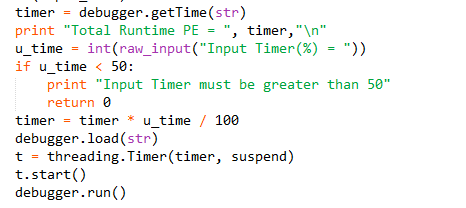
. 

**Gambar 5.3 Implementasi getAddress()**

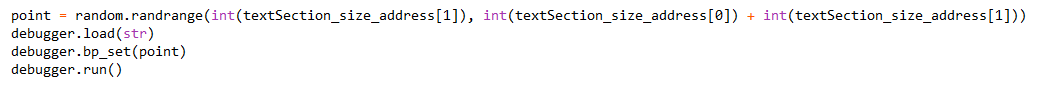
### **Implementasi untuk Mengambil *Snapshot* *Text Section***

Pada implementasi ini, ada 2 cara yang digunakan untuk mengambil *snapshot text section*. *User* dapat memilih salah satu dari kedua cara tersebut. Cara pertama dengan menggunakan *timer* dan cara yang kedua dengan menggunakan *breakpoint* yang di-*set* secara random. Jika *user* memilih menggunakan *timer*, maka *user* juga diminta untuk menentukan berapa persen (%) waktu yang ditentukan untuk mengambil *snapshot* *memory* dari total waktu eksekusi semua isi *text section*.

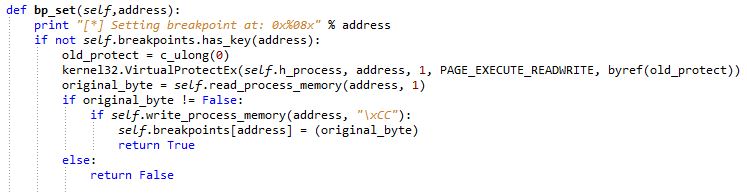
Jika *user* telah mengisi berapa persen waktu yang dibutuhkan untuk mengambil *snapshot text section* pada waktu yang ditentukan kemudian *debugger* akan di suspend() atau diberhentikan sementara sehingga *snapshot text section* dapat diambil. Jika *user* memilih menggunakan *breakpoint*, maka akan dipilih secara acak alamat pada *text section* sehingga jika sudah melalui alamat tersebut maka akan diambil *snapshot* sampai alamat tersebut dengan memanggil fungsi bp\_set(). Untuk kode program pada proses *timer* dapat dilihat pada Gambar 5.4 dan untuk kode program untuk set breakpoint secara random dapat dilihat pada Gambar 5.5.



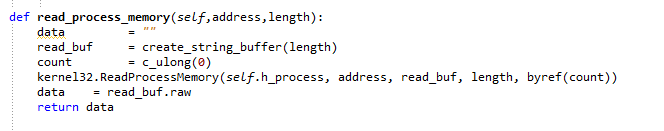
**Gambar 5.4 Implementasi Teknik Timer**



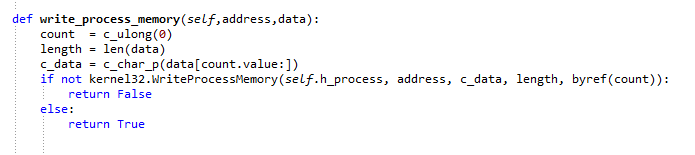
**Gambar 5.5 Implementasi Teknik Set Breakpoint dengan Random**

Pada *Gambar 5.5* dapat dilihat bahwa teknik *set* *breakpoint* menggunakan fungsi bp\_set() yang digunakan untuk men-*set breakpoint* pada awal dari *text section*. Kode program untuk fungsi ini dapat dilihat pada Gambar 5.6.

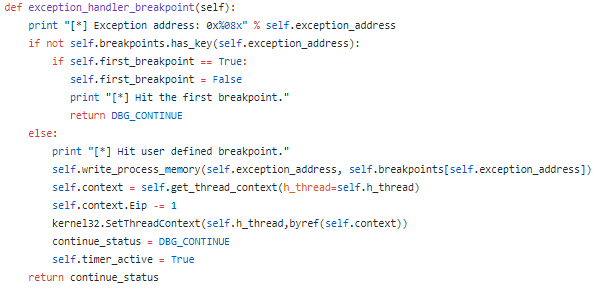
**Gambar 5.6 Implementasi bp\_set**

Pada kode program yang digunakan untuk menentukan breakpoint, menggunakan fungsi read\_process\_memory() dan write\_process\_memory() untuk membaca dan menulis 0xCC pada memori. Kode program fungsi read\_process memory() dan write\_process\_memory()dapat dilihat pada Gambar 5.7 dan Gambar 5.8 pada halaman selanjutnya.

**Gambar 5.7 Implementasi Read Process Memory**



**Gambar 5.8 Implementasi write\_process\_memory**

Apabila *exception* terjadi ketika *debugger* mengakses *text section* maka sistem akan menanganinya dengan menggunakan fungsi *debugger* berupa exeception\_handler\_breakpoint(). Kode program fungsi ini dapat dilihat pada Gambar 5.9.

# **Gambar 5.9 Implementasi exception\_handler\_breakpoint()**

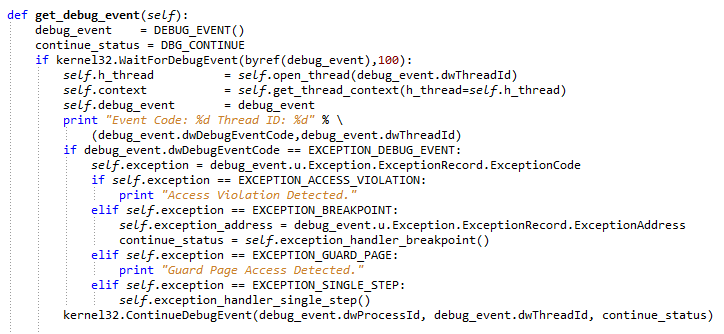
#### **Memanggil Fungsi Run**

Proses ini dilakukan pada teknik *timer* dan *set breakpoint* secara random. Pada implementasi ini berguna untuk mengeksekusi kode pada *debugger*. Pada proses ini menggunakan fungsi debugger\_active dan get\_debug\_event. *Debugger active* ini menandakan *debugger* aktif, dan jika *debugger* aktif maka akan dipanggil fungsi get\_debug\_event(). Kode program pada proses ini dapat dilihat pada Gambar 5.10 pada halaman selanjutnya.



**Gambar 5.10 Implementasi fungsi run()**

Pada fungsi get\_debug\_event() ini berisi fungsi WaitForDebugEvent dan ContinueDebugEvent. Proses akan mengeksekusi tiap-tiap instruksi dan memeriksa apakah instruksi mengandung *exception* atau tidak. Jika mengandung *exception* maka akan ditangani sesuai dengan kondisi *exception* yang ada misalnya exception\_breakpoint dengan menangani alamat *exception-*nya. Jika tidak mengandung *exception* maka akan dijalankan fungsi ContinueDebugEvent. Kode program untuk fungsi ContinueDebugEvent dapat dilihat pada Gambar 5.11.



**Gambar 5.11 Implementasi get\_debug\_event()**

Jika proses *timer* dan *set breakpoint* *random* telah selesai maka akan dipanggil fungsi suspend() yang digunakan untuk menghentikan *debugger* sementara sehingga *snapshot* memori dapat diambil untuk digunakan pada proses selanjutnya. Kode program untuk fungsi suspend() dapat dilihat pada Gambar 5.12.

**

**Gambar 5.12 Implementasi suspend()**

### **Mengambil *Signature Malware* dari *Database***

Proses ini digunakan untuk mengambil *signature* *malware* pada *database*, sehingga *signature* dapat digunakan pada algoritma selanjutnya untuk melakukan proses *string matching*. Kode program pada proses ini dapat dilihat pada Gambar 5.13.

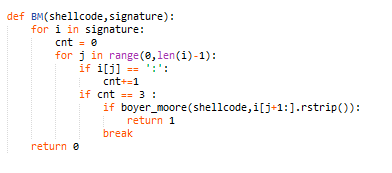


**Gambar 5.13 Implementasi getFromDatabase()**

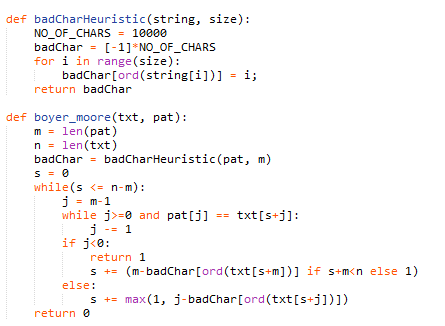
### **5.2.5 Proses *String Matching***

Proses ini digunakan untuk mencari *signature malware* pada *database* apakah sama dengan *snapshot text section* yang telah diambil sebelumnya. Ada beberapa algoritma yang dapat digunakan pada proses *string matching*, seperti: Boyer-Moore, Rabin-Karp, dan Suffix Tree. *User* dapat memilih satu dari ketiga algoritma diatas, yang digunakan untuk proses *string matching*. *Signature malware* dibandingkan dengan *snapshot text section* dalam heksadesimal*.* Jika *signature malware* sama dengan *snapshot text section*, maka dapat dipastikan bahwa *file executable* mengandung *malware*. Sebaliknya jika tidak sama maka *file executable* belum dapat dipastikan mengandung *malware* atau tidak.

#### **5.2.5.1 Algoritma Boyer Moore**

Algoritma ini menggunakan pendekatan *bad character*. Pencarian pada algoritma ini, mengecek kecocokkan per karakter antara *signature malware* dan *shellcode text section*. Jika tidak cocok maka akan dilakukan pergeseran karakter dengan metode *bad character* dan melakukan pencocokan kembali. Kode program pada algoritma Boyer- Moore dapat dilihat pada Gambar 5.14 .

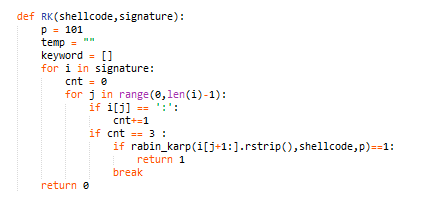
**Gambar 5.14 Implementasi Boyer-Moore**

Untuk kode program untuk fungsi Boyer-Moore dengan pendekatan *bad character* dapat dilihat pada Gambar 5.15.

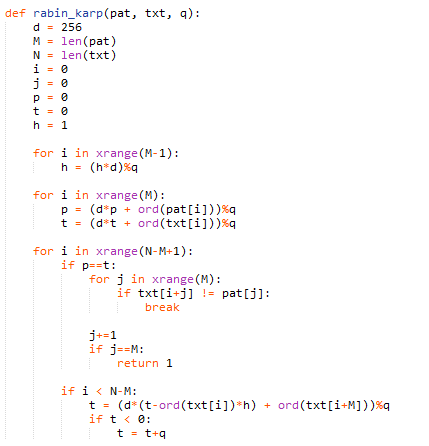
**Gambar 5.15 Implementasi Boyer-Moore dengan pendekatan Bad Character Heuristic**

#### **5.2.5.2 Algoritma Rabin-Karp**

Pada algoritma ini, memperhitungkan panjang dari *shellcode text section* dan *signature* yang akan dibandingkan. Dimana *shellcode text section* dibuah menggunakan *hash value* kemudian dibandingkan dengan *signature*. Jika pada saat membandingkan, *shellcode text section* tidak sama dengan *signature* maka akan di *swap* satu karakter dari *signature* dan dibandingkan lagi, apakah sama dengan *shellcode*. Begitu sampai proses selanjutnya sampai ditemukan kecocokkan antara *shellcode text section* dan *signature malware*. Jika *shellcode text section* dibandingkan dengan semua *signature malware* dan tidak ditemukan kecocokkan maka tidak dapat disimpulkan mengandung *malware* atau tidak. Kode program algoritma Rabin-Karp yang digunakan untuk mencari *shellcode text section* pada *signature malware* dapat dilihat pada Gambar 5.16 dihalaman berikutnya dan implementasi pendefenisian algoritma Rabin-Karp dapat dilihat pada Gambar 5.17 dihalaman selanjutnya.

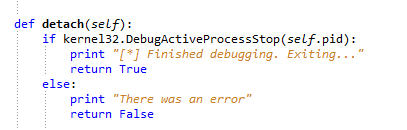


**Gambar 5.16 Implementasi Algoritma Rabin Karp**



**Gambar 5.17 Implementasi Pendefenisian Fungsi Rabin Karp**

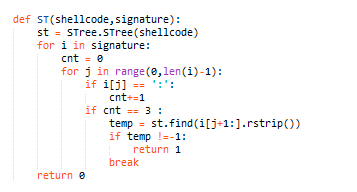
Jika proses *string matching* telah selesai, akan dipanggil fungsi detach() untuk mengakhiri proses *debugging.* Kode program fungsi ini dapat dilihat pada Gambar 5.18.



**Gambar 5.18 Implementasi detach()**

#### **5.2.5.3 Suffix Tree**

Pada algorima ini, menggunakan *tree* dimana *node-node-*nya merupakan bagian *suffix* dari *signature malware* yang diambil dari *database*. Kode program pada algoritma *suffix tree* dapat dilihat pada Gambar 5.19.



**Gambar 5.19 Implementasi Algoritma *Suffix Tree***

# **EKSPERIMEN DAN PEMBAHASAN**

Pada bab ini dijelaskan mengenai tujuan pengujian dan hasil analisis dari hasil pengujian yang diperoleh melalui eksperimen terhadap *malware* yang pada *file executable*.

## **6.1 Pengujian**

Pada sub bab ini akan dijelaskan pengujian yang dilakukan terhadap *custom debugger* pendeteksi *malware* pada *file executable* yang dikembangkan oleh penelitian. Tujuan pengujian adalah untuk membuktikan bahwa *debugger* yang diekstensi telah dapat menghentikan piranti lunak yang dianalisis pada titik perberhentian yang diharapkan serta mampu melakukan *string* *matching* untuk mencari apakah *signature* jahat ada atau tidak pada *file* *executable* yang dianalisis. Dengan kata lain penelitian ingin membuktikan bahwa *custom* *debugger* yang telah diekstensi telah mampu mengatasi kelemahan dari analisis statis. Untuk keperluan pembuktian ini maka setidaknya dibutuhkan jenis *malware* yang belum diobfuskasi, yang telah diobfuskasi dan sebuah *detector* yang bekerja dengan metode analisis statis. *Detector* analisis statis yang kami gunakan adalah sebuah antivirus AVG versi 9.0. AVG versi ini kami gunakan karena diketahui versi tersebut hanya menerapkan analisis statis dalam pendeteksian piranti lunakan jahat tetapi AVG versi sekarang telah menggunakan analisis dinamik seperti *sanboxing*. Oleh sebab itu, perlu dipahami bahwa penelitian ini tidak meng-*out* *perform* antivirus. Selain itu, seperti yang disampaikan sebelumnya pada bab 1.4, *custom* *debugger* yang dibangun belum menerapkan teknik penghindaran deteksi dinamik sehingga bisa saja virus generasi baru tidak akan mampu dideteksi oleh *custom* *debugger* yang dikembangkan. Kami akan mengumpulkan *malware* dimana tahun penemuan yang beragam dari tahun 2010 – 2017.

### **6.1.1 Tujuan Pengujian**

Adapun tujuan dilakukannya pengujian adalah sebagai berikut:

1. Untuk melihat apakah *custom debugger* yang dikembangkan dapat mendeteksi *malware* pada memori *file executable*.
2. Untuk melihat hasil pendeteksian *malware* dengan mendeteksi memori dari *file executable* tersebut.
3. Untuk melihat perbandingan menggunakan *debugger* yang dikembangkan dan antivirus yang telah ada.

### **6.1.2 Persiapan Pengujian**

Persiapan pengujian mencakup objek yang diuji yaitu *file executable*, antivirus, lingkungan pengujian dan skenario pengujian.

#### **6.1.2.1 File Executable**

Pada bagian ini, peneliti akan menggunakan beberapa objek sebanyak 30 *file executable* yang akan diuji. Ada *file executable* yang mengandung *malware* dan yang tidak mengandung *malware*. *File executable* yang mengandung *malware* juga yang diuji ada 2 yaitu telah melalui teknik obfuskasi dengan cara XOR *encoding* dan *malware* tidak melalui teknik obfuskasi. Objek-objek tersebut akan diuji pada custom *debugger* dan antivirus AVG versi lama.

#### **6.1.2.2 Lingkungan Pengujian**

Lingkungan pengujian *custom debugger* sama dengan lingkungan implementasi dengan tambahan antivirus AVG sebagai perbandingan *custom debugger* yang dikembangkan.

#### **6.1.2.3 Skenario Pengujian**

Skenario pengujian yang dilakukan pada penelitian ini adalah sebagai berikut:

1. Mempersiapkan antivirus AVG versi 9.0 sebagai perbandingan dengan *debugger* yang dikembangkan.
2. Mempersiapkan *file executable* sebagai objek yang akan diuji.
3. *Debugger* dijalankan dengan mendeteksi *file executable* yang mengandung *malware*.
4. Kemudian *file executable* tersebut juga dideteksi dengan menggunakan antivirus AVG versi 9.0.
5. Lihat hasil percobaan dari *custom debugger* dan antivirus AVG versi 9.0.
6. Uji lagi *file executable* lainnya, kemudian lihat hasilnya pada antivirus dan *custom debugger*
7. Kemudian kalkulasi keberhasilan *custom debugger* mendeteksi *malware* dibandingkan antivirus yang ada dan hitung keefektifan *custom debugger* tersebut dari beberapa *file executable* yang diuji*.*

## **Pengujian *Custom Debugger***

*Custom debugger* yang dikembangkan terlebih dahulu diuji fungsionalitasnya untuk memastikan *custom debugger* tersebut telah bekerja sesuai dengan algoritma yang telah dirancang sebelumnya. Hasil pengujian *custom debugger* yang telah dikembangkan dideskripsikan dalam bentuk tabel-tabel:

1. Pengujian untuk *file executable* yang diduga *malware* dapat dilihat pada Tabel 6.1.

**Tabel 6.1 Use Case Scenario Pengujian File yang Diduga Malware**

|  |  |
| --- | --- |
| Nama Kasus Uji | Pengujian *file executable* yang diduga *malware* |
| Tujuan | Untuk melihat hasil penilaian yang diperoleh apabila *file executable* yang disediakan merupakan *malware* |
| Deskripsi | Pengujian dilakukan terhadap *file executable* yang telah diduga sebagai *malware* |
| Kondisi Awal | - |
| Skenario Uji | |
| 1. Memberi masukan berupa *path* *file executable* 2. Melihat status *malware* | |
| Kriteria Evaluasi Hasil | |
| *Custom debugger* menghasilkan status pendeteksian berupa ’detected’ | |
| Kasus dan Hasil Uji | |

|  |  |  |  |
| --- | --- | --- | --- |
| Data Masukan | Yang Diharapkan | Pengamatan | Kesimpulan |
|  | *Custom debugger* menghasilkan status pendeteksian berupa ‘detected’ | *Custom debugger* menghasilkan status pendeteksian berupa ‘detected’ | Nilai yang dihasilkan *custom debugger* pada *file executable* yang diduga *malware* sudah baik. |

|  |
| --- |
| Catatan : |
|  |

1. Pengujian untuk *file executable* yang tidak dapat disimpulkan sebagai *malware* dapat dilihat pada Tabel 6.2.

**Tabel 6.2 Use Case Scenario Pengujian File Tidak Dapat Disimpulkan Sebagai Malware**

|  |  |
| --- | --- |
| Nama Kasus Uji | Pengujian *file executable* yang diduga tidak dapat disimpulkan *malware* |
| Tujuan | Untuk melihat hasil penilaian yang diperoleh apabila *file executable* yang disediakan merupakan *malware* |
| Deskripsi | Pengujian dilakukan terhadap *file executable* yang tidak dapat disimpulkan mengandung *malware* atau tidak. |
| Kondisi Awal | - |
| Skenario Uji | |
| 1. Memberi masukan berupa path *file executable* 2. Melihat status *malware* | |
| Kriteria Evaluasi Hasil | |
| *Custom debugger* menghasilkan status pendeteksian berupa ’inconclusive’ | |
| Kasus dan Hasil Uji | |

|  |  |  |  |
| --- | --- | --- | --- |
| Data Masukan | Yang Diharapkan | Pengamatan | Kesimpulan |
|  | *Custom debugger* menghasilkan status pendeteksian berupa ‘*inconclusive’* | *Custom debugger* menghasilkan status pendeteksian berupa ‘*inconclusive’* | Nilai yang dihasilkan *custom debugger* pada *file executable* yang tidak dapat disimpulkan mengandung *malware* sudah baik. |

|  |
| --- |
| Catatan |
|  |

## **Percobaan**

Pada percobaan ini, data percobaan yang digunakan yaitu sebanyak 30 *file executable* yang terdiri dari 28 *file executable* berupa *malware* dan 2 *file executable* bukan *malware*. *File executable* diuji pada *debugger* yang mengandung 100.600 *signature malware* yang diambil dari clamAV. Berikut adalah hasil pengujian yang dilakukan pada *file executable* baik menggunakan *debugger* yang dikembangkan dan antivirus AVG versi 9.0 yang dapat dilihat pada Tabel 6.3. Keterangan *file executable* yang digunakan sebagai data percobaan akan dituliskan dalam Lampiran I pada halaman selanjutnya.

**Tabel 6.3 Percobaan File Executable**

| No. | Nama *File* *executable* | *Debugger* | | | AVG antivirus | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Malware* | Tidak dapat disimpulkan | Kesesuaian | *Malware* | Tidak dapat disimpulkan | Kesesuaian |
| 1 | Variant.Kazy | - | Ya | Tidak Sesuai | Ya | - | Sesuai |
| 2 | Win32\_Alina\_3\_4.exe | - | Ya | Tidak Sesuai | Ya | - | Sesuai |
| 3 | CryptoLocker.exe | Ya | - | Sesuai | - | Ya | Tidak Sesuai |
| 4 | Nivdort | Ya | - | Sesuai | - | Ya | Tidak Sesuai |
| 5 | TrojanWin32.Duqu.Stuxnet | Ya | - | Sesuai | - | Ya | Tidak Sesuai |
| 6 | PEview.exe | - | Ya | Sesuai | - | Ya | Sesuai |
| 7 | Hello\_world.exe | - | Ya | Sesuai | - | Ya | Sesuai |
| 8 | Win32\_Infostealer\_Dexter.exe | Ya | - | Sesuai | - | Ya | Tidak Sesuai |
| 9 | Artemis.exe | - | Ya | Tidak Sesuai | - | Ya | Tidak Sesuai |
| 10 | njRAT.exe | - | Ya | Tidak Sesuai | - | Ya | Tidak Sesuai |
| 11 | Proteus.exe | - | Ya | Tidak Sesuai | - | Ya | Tidak Sesuai |
| 12 | Ransomware\_Cerber.exe | - | Ya | Tidak Sesuai | - | Ya | Tidak Sesuai |
| 13 | Ransomware\_Mamba.exe | - | Ya | Tidak Sesuai | - | Ya | Tidak Sesuai |
| 14 | Ransomware\_Petrwrap.exe | - | Ya | Tidak Sesuai | - | Ya | Tidak Sesuai |
| 15 | Ransomware\_Unnamed\_0.exe | - | Ya | Tidak Sesuai | - | Ya | Tidak Sesuai |
| 16 | zeus\_banking.exe | Ya | - | Sesuai | - | Ya | Tidak Sesuai |
| 17 | Win32\_Unclassified.exe | - | Ya | Tidak Sesuai | - | Ya | Tidak Sesuai |
| 18 | CryptoLocker2.exe | Ya | - | Sesuai | - | Ya | Tidak Sesuai |
| 19 | Nivdort2.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 20 | illusion\_bot | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 21 | Kelihos.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 22 | Rustock\_J.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 23 | Rustock\_23.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 24 | Trojan\_asprox.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 25 | Trojan\_Destover\_Sony.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 26 | Trojan\_kovter.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 27 | AryanRatClient.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 28 | Trojaned.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 29 | Trojan\_encoded.exe | Ya | - | Sesuai | - | Ya | Tidak sesuai |
| 30 | Fax | Ya | - | Sesuai | - | Ya | Tidak sesuai |

Hasil pengujian menunjukkan seberapa banyak *file executable* yang mengandung *malware* dapat dideteksi oleh *debugger* dan perbandingannya dengan antivirus AVG versi 9.0.

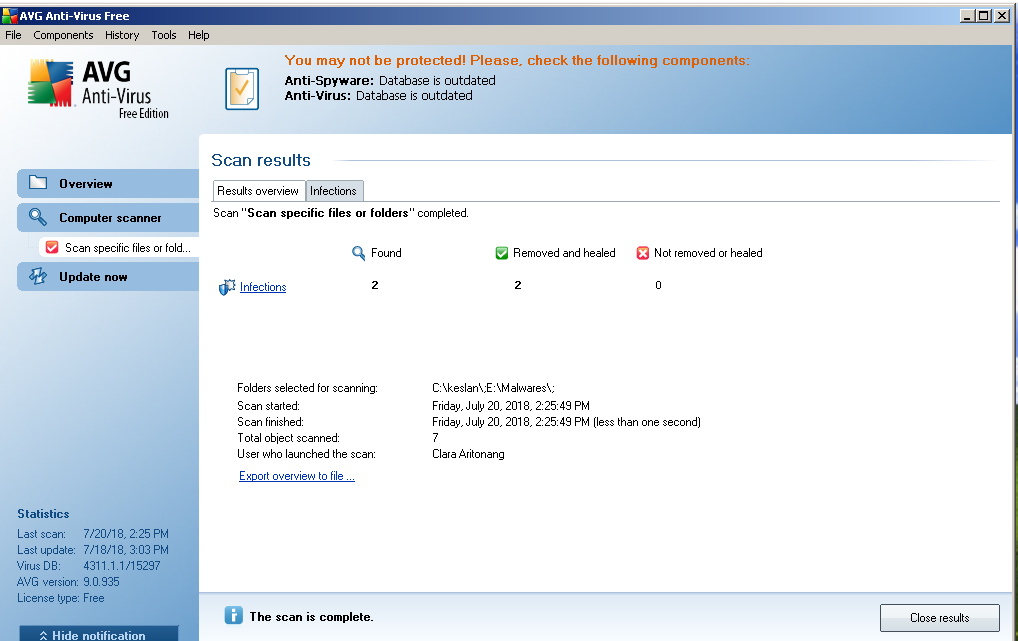
Berdasarkan hasil pengujian pada *debugger* dapat dilihat bahwa,

* Jumlah *file executable* yang dapat dideteksi *debugger* = 20
* Jumlah *file executable* yang tidak dapat dideteksi *debugger =* 10
* Jumlah semua *file executable* yang digunakan = 30

Perhitungan akurasi dari *debugger* dalam mendeteksi *malware* yaitu:

**( 6.1 )**

Berdasarkan hasil percobaan, ada beberapa *file executable* yang tidak dapat dideteksi oleh antivirus AVG dan *debugger* yang dikembangkan oleh peneliti. Menurut peneliti, *debugger* tidak dapat mendeteksi *file executable* tersebut dikarenakan *file* tersebut menggunakan teknik *anti*-*debugger* dimana jika *file* tersebut dieksekusi pada *debugger* maka *malware* akandisembunyikan atau beberapa teknik canggih lainnya yang dibuat oleh *attacker* sehingga *malware* tidak dapat dideteksi di *debugger* atau di anti virus lainnya. Antivirus AVG juga tidak dapat mendeteksi beberapa *file* dikarenakan antivirus tersebut versi lama yang belum mengatasi *malware* yang disimpan dalam memori atau melakukan teknik obfuskasi atau teknik – teknik lain untuk mencegah *malware* tersebut dideteksi. Beberapa *file executable* yang tidak dapat dideteksi oleh AVG disimpan pada folder “C:\keslan\E:\Dyne\Win32.infostealer.Dexter\”. *Screenshot* dari hasil pendeteksian *malware* pada folder tersebut dapat dilihat pada Gambar 6.1 dihalaman berikutnya.



**Gambar 6.1 Screenshot Pendeteksian AVG**

Berdasarkan hasil pengujian pada antivirus AVG dapat dilihat bahwa,

* Jumlah *file executable* yang dapat dideteksi antivirus AVG = 4
* Jumlah *file executable* yang tidak dapat dideteksi antivirus AVG *=* 26
* Jumlah semua *file executable* yang digunakan = 30

Perhitungan akurasi dari antivirus AVG versi 9.0 dalam mendeteksi *malware* yaitu:

**( 6.2 )**

Dari kedua hasil perhitungan dibandingkan hasil akurasi pendeteksian *malware* menggunakan *debugger* lebih besar daripada menggunakan antivirus AVG versi 9.0. Hal ini menjadi perbandingan bagi peneliti, dimana pendeteksian dengan analisis dinamik lebih baik daripada analisis statis karena dari sejumlah virus yang diuji di antivirus AVG dan *debugger* ditemukan lebih banyak virus yang dapat dideteksi oleh *debugger* dibandingkan dengan antivirus AVG karena pendeteksian *malware* menggunakan *debugger* mampu mendapatkan *signature malware* dengan metode *in memory scanning* sedangkan antivirus memeriksa *file executable* berdasarkan *behavior*-nya.

## **Pembahasan**

Pada subbab ini dibahas mengenai analisis terhadap hasil pengujian dan perbandingan akurasi hasil penilaian menggunakan antivirus AVG versi 9.0 dan *custom debugger* yang telah dilakukan.

### ***Custom Debugger***

*Custom debugger* yang dibangun oleh peneliti menggunakan fungsi – fungsi Windows yang dituangkan dalam kodingan dengan bahasa pemograman Python. Pada pembangunan *debugger* ini, *file executable* di-*load* dengan fungsi *create process* sehingga *debugger* dapat mengakses semua isi memori. Hal ini juga terjadi jika *user* meng-*input path file* dari *file* tersebut. Kemudian *debugger* yang dibangun menggunakan fungsi PE yang digunakan untuk mengetahui letak *text section* dimana *text section* berisi kode program dalam *memory file*. Dengan fungsi ini, peneliti lebih mudah mengetahui letak *text section* dan *size*-nya. Fungsi PE ini merupakan fungsi Windows. Sehingga dengan fungsi tersebut, peneliti dapat mengambil dan mengeksekusi *text section* dari *file* tersebut. Selanjutnya untuk menghentikan *debugger*, peneliti mengimplementasikan 2 cara, baik dengan cara *timer* dan *set breakpoint* secara acak.

* *Timer*.

*User* diberikan kebebasan untuk menentukan seberapa lama waktu yang diinginkan untuk mendeteksi *text section*, maksudnya *user* diminta untuk meng-*input* berapa persen (%) dari total waktu keseluruhan untuk mendeteksi *malware*. Jika user memilih 50% maka *debugger* hanya memeriksa *text section* sampai 50% dari total waktu, itu berarti tidak semua *text section* diperiksa. Hal ini dapat memungkinkan, *malware* tidak terdeteksi karena peneliti atau *user* tidak dapat memprediksi atau mengetahui letak *malware* pada *text section*. Dengan kata lain, jika *user* menginput waktu dibawah 100 % berarti *user* menebak bahwa letak *malware* ada pada waktu yang di-*input*-nya. Jika menggunakan waktu 100 % maka keakuratan *file* tersebut mengandung *malware* atau tidak sangat tinggi karna *debugger* mengecek semua isi *text section* tetapi waktu yang dibutuhkan untuk mengeksekusi semua isi *text section* lama tergantung panjang *text section* dan *signature* yang ada pada *library*.

* *Set breakpoint* secara random

Pada metode ini menggunakan fungsi random() dimana *debugger* akan meletakkan *breakpoint* pada sembarang tempat atau secara *random* dan pada saat mengeksekusi *malware*, *debugger* akan berhenti pada *address* yang telah di *set* sebelumnya. Ini merupakan cara tebak-tebakan karena *breakpoint* di-*set* disembarang *address* pada *text section*, bisa saja pertengahan *text section*, diakhir *text section* atau 2 *address* setelah awal dari *text sectio*n. Keakuratan dalam penggunaan metode ini tidak dapat dipastikan.

Setelah menggunakan cara *timer* dan *set breakpoint*, maka *debugger* akan memanggil fungsi suspend() untuk menghentikan *debugger* sementara sehingga dapat mengambil *snapshot* dari *text section*. Setelah didapat *snapshot*-nya, maka akan digunakan algoritma *string matching* untuk *snapshot text section* yang didapat dan *signature malware* dari ClamAV. Ada beberapa algoritma yang digunakan oleh peneliti, yaitu: Boyer-Moore, Suffix Tree dan Rabin-Karp.

#### **Boyer-Moore**

Algoritma ini menggunakan pendekatan *bad character*, dimana *shellcode text section malware* dicari satu – satu pada *library signature malware*. Untuk setiap *signature*, jika tidak memenuhi maka akan di *swap* sesuai aturan *bad character*. Hal ini membuat pencarian *malware* pada *file executable* tidak cepat. Dari hasil penelitian, algoritma ini hampir sama dengan algoritma *suffix tree*. Tetapi algoritma *Suffix tree* lebih cepat 1 menit atau 2 menit.

* + - 1. *Suffix Tree*

Algoritma ini merupakan algoritma yang lebih cepat dari kedua algoritma lainnya. Hal ini dikarenakan, pada algoritma ini mengunakan *tree*. Dimana *signature malware* dibuat dalam *tree* dan dikompresi. Pencarian dengan *tree* dilakukan secara paralel sehingga mempercepat pencarian. Dari percobaan yang dilakukan oleh peneliti, algoritma ini lebih cepat dibandingkan kedua algoritma lainnya namun memori yang digunakan pada algoritma ini lebih besar.

* + - 1. Rabin-Karp

Pada algoritma ini, sangat memperhitungkan panjang dari *shellcode text section* yang akan dicari pada *library signature malware* dan menggunakan *hash value* pada *shellcode text section*. *Hash value* tersebut dibandingkan dengan *library malware*, jika panjang *shellcode text section* 42 karakter dan panjang *signature malware* hanya 30 karakter maka *signature* tersebut dilewatkan dan diganti dengan *signature* berikutnya. Sebaliknya jika *shellcode text section* 10 karakter dan panjang *library signature malware* 100 maka *shellcode* akan dicek dari karakter pertama ke karakter ke 10. Jika tidak cocok maka akan di *swap* 1 karakter dan dibandingkan lagi. Jika tidak ditemukan pada satu *signature*, maka akan dicoba ke *signature* selanjutnya dengan proses yang sama. Dari hasil penelitian, algoritma ini terbilang lama dibanding algoritma lainnya.

Dari ketiga algoritma tersebut, *suffix tree* merupakan algoritma yang lebih cepat melakukan proses *string matching* namun dengan algoritma ini, *memory space* yang digunakan lebih banyak dari algoritma lainnya seperti: Rabin karp dan Boyer moore. Waktu yang digunakan pada proses *string matching* dipengaruhi juga dengan panjang dari *text section* dan banyaknya *signature malware* yang digunakan sebagai *library* untuk menentukan *file executable* tersebut *malware* atau tidak.

* + 1. Antivirus AVG

Pada penelitian ini, peneliti menggunakan AVG versi 9.0 sebagai perbandingan untuk mendeteksi *malware* dengan cara statis (AVG versi 9.0) dan cara dinamik (*custom debugger* yang dikembangkan peneliti). Anti virus ini memeriksa *malware* berdasarkan *behavior* dari *file* tersebut tanpa memeriksa memori atau isi dari *file* tersebut. Sehingga banyak antivirus lama yang tidak dapat mendeteksi *malware*, karena *malware* telah menggunakan teknik-teknik penanganan *malware* seperti mengobfuskasi atau mengenkripsi *file* tersebut. Dari percobaan yang dibuat, ada beberapa *file* yang tidak dapat dideteksi oleh *malware*. Dalam hal ini, peneliti tidak meneliti alasan kenapa antivirus AVG tidak dapat mendeteksi. Peneliti dapat memastikan, *malware* pada *file* tersebut melakukan penanganan terhadap pendeteksian *malware* atau meletakkan *malware* pada *text section file*. Pada penelitian ini, tidak untuk mengalahkan atau menyaingi antivirus (AVG versi 9.0) tetapi antivirus versi 9.0 ini masih menggunakan pengecekan *malware* secara statik sehingga peneliti menggunakan AVG versi 9.0 sebagai pembanding dalam pengecekan *malware* baik secara statik atau dinamik.

* + 1. Hasil Percobaan

Dari hasil percobaan pada subbab 6.3, beberapa *file* yang mengandung *malware* atau yang tidak mengandung *malware* dapat dideteksi oleh *debugger*, tetapi ada sebagaian *file* yang mengandung *malware* tidak dapat dideteksi oleh AVG. gambar dibawah ini merupakan beberapa *screenshot* pendeteksian *malware* dengan *debugger* dengan algoritma *Suffix tree*. Hasil yang diperoleh juga menunjukkan perolehan kompleksitas ruang dan waktu beberapa *file* sampel *malware*. Kompleksitas dapat kita bagi menjadi dua: kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu adalah waktu yang dibutuhkan oleh komputer untuk menjalankan sebuah algoritma sampai selesai. Kompleksitas ruang adalah memori yang dibutuhkan oleh komputer untuk menjalankan sebuah algoritma sampai selesai.

1. Pendeteksian *malware* pada *file* fax.exe dengan menggunakan *debugger* dapat dilihat pada Gambar 6.2.

E : \VirtualShare\TA\src>python my\_test.py

Enter the name of your text file – please use / backslash when typing in directory path: fax.exe

1. Timer
2. Breakpoint

Input = 1

[\*] We have successfully launched the process!

[\*] The Process ID I have is: 4600

Total Runtime PE = 0.131999969482

Input Timer(%) = 10

[\*] We have successfully launched the process!

[\*] The Process ID I have is: 8792

start Text section = 4198400

end Text section = 4493312

Choose Algorithm :

1. Suffix Tree
2. Boyer Moore
3. Rabin Karp
4. Aho Corasick

Input: 1

Malware

Total RunTime Search = 21.0119998455

**Gambar 6.2 Screenshot Pendeteksian fax.exe**

1. Pendeteksian *malware* pada *file* trojaned.exe dengan menggunakan *debugger* dapat dilihat pada Gambar 6.3.

E : \VirtualShare\TA\src>python my\_test.py

Enter the name of your text file – please use / backslash when typing in directory path: trojaned.exe

1. Timer
2. Breakpoint

Input = 1

[\*] We have successfully launched the process!

[\*] The Process ID I have is: 4600

Total Runtime PE = 0.616000175476

Input Timer(%) = 100

[\*] We have successfully launched the process!

[\*] The Process ID I have is: 4092

Choose Algorithm :

1. Suffix Tree
2. Boyer Moore
3. Rabin Karp
4. Aho Corasick

Input: 1

Malware

Total RunTime Search = 6.00099992752

**Gambar 6.3 Screenshot Pendeteksian trojaned.exe**

1. Pendeteksian *malware* pada *file* win33.exe dengan menggunakan *debugger* dapat dilihat pada Gambar 6.4.

**Gambar6.4 *Screenshot* Pendeteksianwin33*.*exe**

E : \VirtualShare\TA\src>python my\_test.py

Enter the name of your text file – please use / backslash when typing in directory path: win33.exe

1. Timer
2. Breakpoint

Input = 1

[\*] We have successfully launched the process!

[\*] The Process ID I have is: 2476

Total Runtime PE = 0.0199999809265

Input Timer(%) = 100

[\*] We have successfully launched the process!

[\*] The Process ID I have is: 11084

Choose Algorithm :

1. Suffix Tree
2. Boyer Moore
3. Rabin Karp
4. Aho Corasick

Input: 1

Malware

Total RunTime Search = 6.05500006676

1. Pendeteksian *malware* pada *file* trojan\_encoded.exe dengan menggunakan *debugger* dapat dilihat pada Gambar 6.5.

E : \VirtualShare\TA\src>python my\_test.py

Enter the name of your text file – please use / backslash when typing in directory path: Trojan\_encoded.exe

1. Timer
2. Breakpoint

Input = 1

[\*] We have successfully launched the process!

[\*] The Process ID I have is: 10640

Total Runtime PE = 17.1949999332

Input Timer(%) = 100

[\*] We have successfully launched the process!

[\*] The Process ID I have is: 5428

start Text section = 4198400

end Text section = 4288512

Choose Algorithm :

1. Suffix Tree
2. Boyer Moore
3. Rabin Karp
4. Aho Corasick

Input: 1

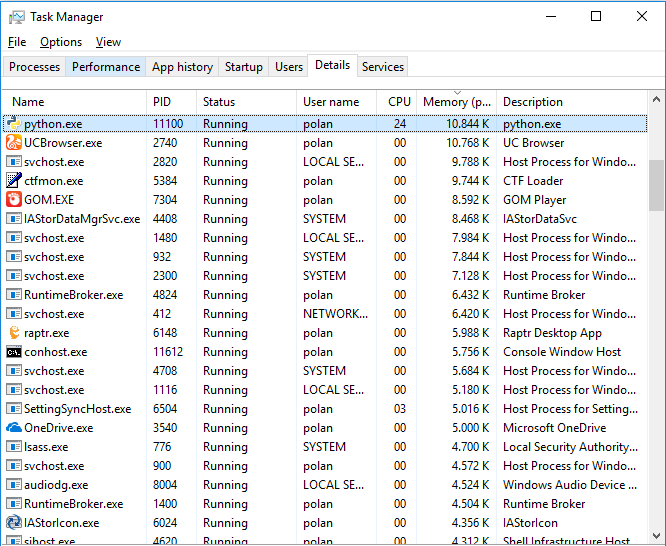
Malware

Total RunTime Search = 13.5670001507

**Gambar 6.5 Screenshot Pendeteksian trojan\_encoded.exe**

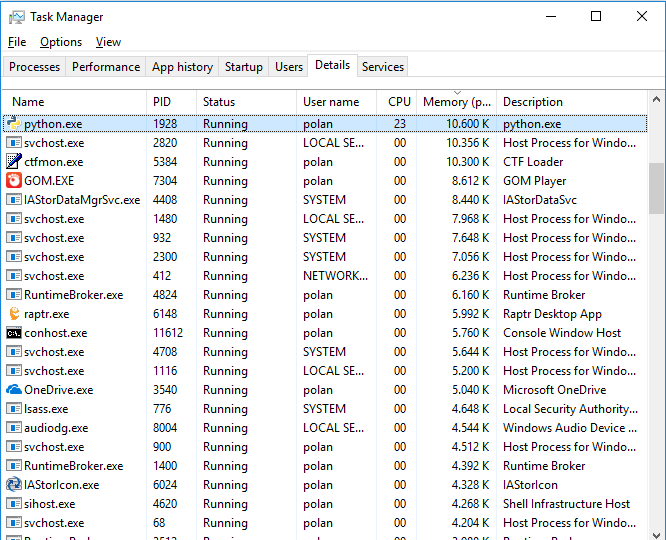
Perolehan memori yang digunakan pada proses pencocokan *string* bergantung pada konsep algoritma itu sendiri.

1. Algoritma Boyer-Moore



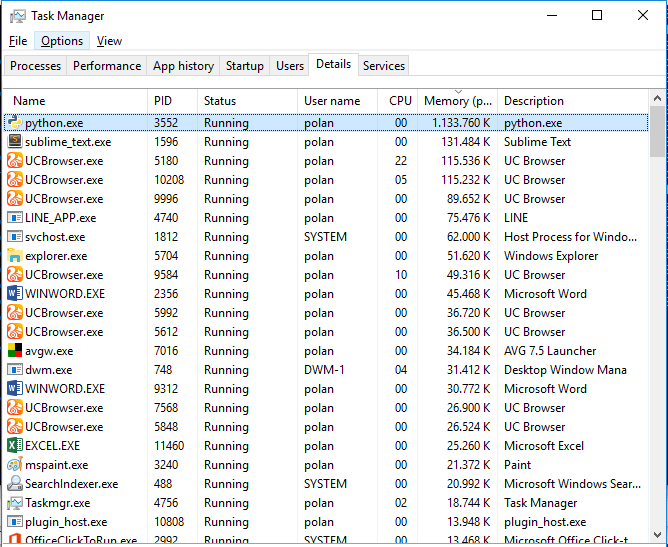
**Gambar 6.6 Hasil Memori Boyer-Moore**

1. Algoritma Rabin-Karp



**Gambar 6.7 Hasil Memori Rabin-Karp**

1. *Suffix Tree*



**Gambar 6.8 Hasil Memori Suffix Tree**

# **BAB 7 KESIMPULAN DAN SARAN**

Pada bab ini dijelaskan kesimpulan dari penelitian pendeteksian *malware* menggunakan metode *memory scanning* serta saran dari penulis terkait pengembangan penelitian yang dapat dilakukan.

## **Kesimpulan**

Penelitan ini telah berhasil memberikan kontribusi terhadap penyelesaian masalah pendeteksian *malware* yang menerapkan teknik analisis dinamik. Berdasarkan hasil implementasi eksperimentasi hasil dan pembahasan dapat disimpulkan keluaran penilaian sebagai berikut:

1. Ada 2 algoritma yang dikembangkan untuk menentukan waktu pengambilan *snapshot* atau isi memori yakni menggunakan metode *timer* dan *breakpoint*. Pada metode *timer*, terlebih dahulu dilakukan penghitungan total waktu eksekusi dari sebuah *executable*. Setelah itu, memori *snapshot* diambil berdasarkan proporsi waktu eksekusi. Sebaliknya pada metode *breakpoint*, *snapshot* memori dilakukan dengan menentukan titik-titik penghentian program atau *breakpoint* secara acak.
2. Penelitian ini menggunakan Windows *debugger* yang dikembangkan oleh Seitz [3] untuk menerapkan algoritma pengambilan *snapshot* memori. Modifikasi minimal dilakukan pada *debugger* untuk mendukung metode *timer*, sedangkan fungsi-fungsi lain untuk instrumentasi dibuat sebagai modul terpisah yang merupakan ekstensi dari *debugger*.
3. Ada 3 algoritma *string matching* yang diterapkan pada *custom debugger* pada penelitian ini yaitu algoritma Boyer-Moore, algoritma Rabin-Karp, dan Suffix Tree. Penelitian ini melakukan pengembangan algoritma dilakukan dengan mendapatkan penanda *malware* dari piranti lunak dan sekumpulan penanda dari basis data ClamAV untuk dilakukan pencocokan tanda-tanda *malware*. Penanda atau *signature* diubah kebentuk *string* untuk dapat diproses oleh algoritma *string matching*. Berdasarkan eksperimentasi yang dilakukan, sistem yang dikembangkan mampu membedakan piranti lunak jahat dari piranti lunak baik berdasarkan pemindaian memori dan mencocokkannya dengan *signature database*.

## **Saran**

Berdasarkan hasil percobaan yang telah dilakukan dan permasalahan yang ditemukan, berikut adalah saran dari penulis yang dapat dilakukan pada percobaan berikutnya:

1. Mengomptimasi waktu yang digunakan untuk pendeteksian *malware* dan dengan cara yang lebih baik.
2. Meningkatkan kemampuan *debugger* dalam mendeteksi *malware* yang mampu mengatasi teknik-teknik penghindaran *debugger*.
3. Mengintegrasikan sistem *custom debugger* dengan sistem analisis dinamik lainnya seperti Cuckoo Sandbox yang bekerja sebagai *framework* yang menganalisa *behaviour* dari *malware*. Sehingga dapat mendeteksi *malware* yang menggunakan teknik-teknik penghindaran pendeteksian *malware*.
4. Melakukan pengembangan algoritma pendeteksian memori atau algoritma *snapshot* memori sehingga lebih efisien.

# **DAFTAR PUSTAKA**

|  |  |
| --- | --- |
| [1] | J. Edwards, S. Turner and J. Spurlock, "Method and System for Detecting Computer Malwares by Scan of Process Memory After Process Initialization," 2003. [Online]. Available: https://patents.google.com/patent/US20030115479. [Accessed 18 February 2018]. |
| [2] | M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press, Inc., 2012. |
| [3] | J. Seitz, *GRAY HAT PYTHON "Python Programming for Hackers and Reverse Engineers",* San Francisco: No Starch Press, Inc, 2009. |
| [4] | S. Kramer and J. C. Bradfield, "A General Definition of Malware," *J Comput Virol,* vol. 6, pp. 105-114, 9 September 2009. |
| [5] | BPPT CSIRT, "Panduan Penanganan Insiden Malware," 2014. [Online]. Available: https://csirt.bppt.go.id/wp-content/uploads/2014/06/Panduan-malware.pdf. [Accessed 2 Juli 2018]. |
| [6] | J. Aycock, "Definisions and Timeline," in *Computer Viruses and Malware*, Canada, Business Media, LLC, 2006, pp. 11-18. |
| [7] | I. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," in *International Conference on Broadband, Wireless Computing, Communication and Applications*, Korea, 2010. |
| [8] | D. Keragala, "Detecting Malware and Sandbox Evasion Techniques," The SANS Institute, United State, 2016. |
| [9] | M. Pietrek, "Microsoft Developer Network," Microsoft, Maret 1994. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms809762.aspx. |
| [10] | A. Silberschatz, P. B. Galvin and G. Gagne, "Memory Management," in *Operating System Concept 9th Edition*, United States, John Wiley & Sons, Inc., 2013, pp. 351-376. |
| [11] | Microsoft developer Network, "About Processes and Threads," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681917(v=vs.85).aspx. [Accessed 23 April 2018]. |
| [12] | Microsoft, "ReadProcessMemory function," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680553(v=vs.85).aspx. [Accessed 2 May 2018]. |
| [13] | Microsoft, "WriteProcessMemory function," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681674(v=vs.85).aspx. [Accessed 2 May 2018]. |
| [14] | Microsoft Developer Network, "VirtualProtectEx function," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366899(v=vs.85).aspx. [Accessed 23 April 2018]. |
| [15] | Microsoft Developer Network, "VirtualQueryEx function," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366907(v=vs.85).aspx. [Accessed 23 April 2018]. |
| [16] | Microsoft, "Basic Debugging," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms679276(v=vs.85).aspx. [Accessed 2 May 2018]. |
| [17] | Microsoft, "About Basic Debugging," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms679266(v=vs.85).aspx. |
| [18] | R. Kath, "Microsoft Developer Network," Microsoft, 5 November 1992. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms809754.aspx. |
| [19] | Microsoft, "ContinueDebugEvent function," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms679285(v=vs.85).aspx. [Accessed 2 May 2018]. |
| [20] | Microsoft, "DebugActiveProcess function," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms679295(v=vs.85).aspx. [Accessed 2 May` 2018]. |
| [21] | MIcrosoft, "DebugActiveProcessStop function," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms679296(v=vs.85).aspx. [Accessed 2 May 2018]. |
| [22] | Microsoft, "WaitForDebugEvent function," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681423(v=vs.85).aspx. [Accessed 2 May 2018]. |
| [23] | Microsoft, "DEBUG\_EVENT structure," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms679308(v=vs.85).aspx. [Accessed 2 May 2018]. |
| [24] | Microsoft, "Debugging Events," Microsoft Developer Network, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms679302(v=vs.85).aspx. [Accessed 2 May 2018]. |
| [25] | A. Schuster, "Searching for Processes and Threads in Microsoft Windows Memory Dumps," in *Digital Forensic Research Conference*, Bonn, 2006. |
| [26] | K. Ask, "Signature," in *Automatic Malware Signature Generation*, 2006, pp. 34-36. |
| [27] | A. Inc, "Apple Open Source," 09 Desember 2007. [Online]. Available: https://opensource.apple.com/source/clamav/clamav-158/clamav.Bin/clamav-0.98/docs/signatures.pdf. [Accessed 12 Juli 2018]. |
| [28] | R. A. Baeza-Yates, "Algorithms for String Searching: A Survey," *ACM SIGIR Forum,* pp. 34-58, April 1989. |
| [29] | D. E. Knuth, J. E. Morris and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Comput.,* vol. 6, no. 2, pp. 323-350, 1997. |
| [30] | R. M. Karp and M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM Journal of Research and Development,* vol. 32, no. 2, pp. 249-260, 1987. |
| [31] | E. Ukkonen, "On-Line Construction of Suffix Trees," *Algorithmica,* no. 14, pp. 249-260, 1996 August 1995. |
| [32] | SANS, "SANSNewsletter," December 2014. [Online]. Available: https://www.dps.texas.gov/SecurityReview/SANSNewsletters/Ouch2014-12.pdf. [Accessed 2 July 2018]. |
| [33] | B. Min, V. Varadharajan, U. Tupakula and M. Hitchens, "Antivirus Security: Naked During Updates," John Wiley & Sons, Ltd, Sydney, 2013. |
| [34] | M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format," March 1994. |

**Lampiran I: File Executable berisi Malware**

| **No.** | ***File Executable*** | **Tipe *Malware*** | **Nama *Malware*** | **Tahun Penemuan** | **Sumber** |
| --- | --- | --- | --- | --- | --- |
| 1. | Variant.Kazy | Trojan | **Gen.Variant.Kazy or Kazy Trojan** | 2010 | https://www.enigmasoftware.com/kazytrojan-removal/ |
| 2. | Win32\_Alina\_3\_4.exe | Trojan-Spy | Win32.Alinaos | 2013 | http://www.virusradar.com/en/Win32\_Alinaos.B/description |
| 3. | CryptoLocker.exe | Ransomware Trojan | CryptoLocker | 2013 | <https://www.secureworks.com/research/cryptolocker-ransomware> |
| 4. | Nivdort | Virus | TrojanSpy:Win32/Nivdort.CT | 2015 | <https://www.symantec.com/security-center/writeup/2010-071400-3123-99> |
| 5. | TrojanWin32.Duqu.Stuxnet.exe | Trojan | Duqu Stuxnet | 2009 | https://www.symantec.com/content/en/us/enterprise/media/security\_response/whitepapers/w32\_duqu\_the\_precursor\_to\_the\_next\_stuxnet.pdf |
| 6. | Win32\_Infostealer\_Dexter.exe | Trojan | Infostealer.Dexter | 2012 | https://us.norton.com/online-threats/infostealer.dexter-2012-121219-2643-99-writeup.html |
| 7. | Artemis.exe | Trojan | Artemis | 2010 | h[ttps://home.mcafee.com/virusinfo/virusprofile.aspx?key=259866](https://home.mcafee.com/virusinfo/virusprofile.aspx?key=259866) |
| 8. | njRAT.exe | Trojan | njRAT | 2014 | https://home.mcafee.com/virusinfo/virusprofile.aspx?key=8988380 |
| 9. | Proteus.exe | Trojan | Proteus | 2016 | https://www.solvusoft.com/en/malware/trojans/w97m-proteus/ |
| 10. | Ransomware\_Cerber.exe | Ransomware Trojan | Cerber ransomware | 2016 | https://www.symantec.com/security\_response/earthlink\_writeup.jsp?docid=2016-030408-0817-99 |
| 11. | Ransomware\_Mamba.exe | Ransomware Trojan | Mamba ransomware | 2016 | https://cyware.com/news/everything-you-should-know-about-mamba-ransomware-c07891a7 |
| 12. | Ransomware\_Petrwrap.exe | Ransomware Trojan | Petya ransomware | 2016 | https://www.symantec.com/security-center/writeup/2016-032913-4222-99 |
| 13. | Ransomware\_Unnamed\_0.exe | Ransomware Trojan | Tidak diketahui | Tidak diketahui | - |
| 14. | zeus\_banking.exe | Trojan | Zeus | 2007 | https://usa.kaspersky.com/resource-center/threats/zeus-virus |
| 15. | Win32\_Unclassified.exe | Tidak diketahui | Tidak diketahui | Tidak diketahui | - |
| 16. | CryptoLocker2.exe | Ransomware Trojan | Cryptolocker 2.0 | 2013 | https://www.comparitech.com/blog/information-security/the-history-of-ransomware/#gref |
| 17. | Nivdort2.exe | Virus | Nivdort | 2016 | https://www.hkcert.org/my\_url/en/blog/16103102 |
| 18. | illusion\_bot.exe | Bot | IllusionBot | Tidak diketahui | - |
| 19. | Kelihos.exe | Trojan | Win32/Kelihos | 2009 | https://www.virusbulletin.com/conference/vb2011/abstracts/same-botnet-same-guys-new-code/ |
| 20. | Rustock\_N.exe | Trojan | Win32:Rustock-N | 2007 | https://www.solvusoft.com/en/malware/trojans/win32-rustock-n/ |
| 21. | Rustock\_23.exe | Trojan | Win32:Rustock-23 | 2006 | https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32%2FRustock |
| 22. | Trojan\_asprox.exe | Trojan | Asprox | 2007 | https://www.symantec.com/security-center/writeup/2007-060812-4603-99 |
| 23. | Trojan\_Destover\_Sony.exe | Trojan | Destover | 2012 | https://securityaffairs.co/wordpress/42194/malware/destover-malware-analysis.html |
| 24. | AryanRatClient.exe | Trojan | AryanRAT | 2010 | ? |
| 25. | Trojan\_kovter.exe | Trojan | Kotver | 2015 | https://www.symantec.com/security-center/writeup/2015-082817-0932-99 |
| 26. | Trojaned.exe | Trojan | *File executable* modifikasi | Tidak diketahui | - |
| 27. | Trojan\_encoded.exe | Trojan | *File executable* modifikasi | Tidak diketahui | - |
| 28. | Fax.exe | Trojan | Tidak diketahui | Tidak diketahui | - |

1. *Remote Administration Tool or Remote Access Trojan* [↑](#footnote-ref-1)