

# **Universidade Federal do Paraná**

Loirto Alves dos Santos  
Luiz Henrique Pires de Camargo

## **Vírus de computador** **Uma abordagem do código polimórfico**

**Curitiba - PR**

2013

# **Universidade Federal do Paraná**

Loirto Alves dos Santos

Luiz Henrique Pires de Camargo

## **Vírus de computador**

**Uma abordagem do código polimórfico**

Monografia apresentada junto ao curso de Ciência da Computação, do Departamento de Informática, do Setor de Ciências Exatas, como requisito parcial para a obtenção do título de Bacharel.

**Orientador:** Prof. Dr. Bruno Müller Junior

**Curitiba - PR**

2013

*A nossos pais. Sem eles nada disso teria sentido.*

# Agradecimentos

- A Deus
- A nosso esforço e dedicação que, apesar de serem poucos, nos valeram muito.
- Aos professores pela paciência e dedicação

# Resumo

Este trabalho tem por finalidade realizar um estudo sobre alguns algoritmos e técnicas de polimorfismo utilizadas para criar vírus de computador e o quanto elas tornam difícil - e algumas vezes até mesmo impossível - a detecção do código malicioso.

**Palavras-chave:** Vírus, Vírus de computador, Vírus polimórfico, polimorfismo.

# Abstract

This paper aims to conduct a study of some algorithms and techniques used to create polymorphic computer viruses and how they make it difficult - and sometimes even impossible - to detect the malicious code.

**Keywords:** Virus, Computer Virus, Polymorphic virus, Polymorphism.

# Sumário

<b>Agradecimentos</b>	<b>iv</b>
<b>Resumo</b>	<b>v</b>
<b>Abstract</b>	<b>vi</b>
<b>Sumário</b>	<b>vii</b>
<b>1 Introdução</b>	<b>1</b>
<b>2 Revisão bibliográfica</b>	<b>2</b>
2.1 Antivírus . . . . .	2
2.2 História . . . . .	2
2.3 Antivirus e SO . . . . .	2
2.3.1 DOS . . . . .	3
2.3.2 Windows . . . . .	3
2.3.3 Linux . . . . .	3
2.4 Técnicas de detecção . . . . .	4
2.4.1 Virus de pendrive . . . . .	4
2.4.2 Virus de macro . . . . .	4
2.4.3 Virus Polimórficos . . . . .	5

<b>3 Polimorfismo</b>	<b>6</b>
3.1 O que é o polimorfismo?	6
3.1.1 Polimorfismo usado de forma legal	6
3.1.2 Como funciona?	7
<b>4 O vírus polimórfico</b>	<b>14</b>
<b>5 Conclusão</b>	<b>16</b>
<b>Referências Bibliográficas</b>	<b>17</b>
<b>A Estrutura de Arquivos PE e ELF</b>	<b>19</b>
A.1 Arquivo PE	19
A.1.1 Estrutura de arquivo PE.	19
A.1.2 PE - Cabeçalho	20
A.1.3 Tabela de Seções	20
A.1.4 Páginas de imagem	21
A.1.5 Importação	21
A.1.6 Exportação	22
A.1.7 Correção	22
A.1.8 Recursos	22
A.1.9 Debug	22
A.2 Arquivo ELF	22
A.2.1 A estrutura do arquivo ELF	23



## Introdução

Nossa vida moderna é extremamente dependente de computadores: desktops, notebooks, netbooks, PDA, celulares, satélites, veículos, microondas, televisores, gps, bancos, energia elétrica, comunicações ..., enfim, uma gama enorme de exemplos poderiam ser citados. Dentro deste contexto, os vírus de computador (e suas variações) são uma ameaça real à qual todos - direta ou indiretamente - estamos expostos.

## Revisão bibliográfica

### 2.1 Antivírus

Os antivírus<sup>4</sup> são softwares criados para analisar, detectar, eliminar e impedir os vírus informáticos ou ao menos diminuir a intensidade do ataque. Foram criados pela necessidade de que os vírus impedissem a utilização do sistema. Os vírus atuais são mais poderosos, e ainda existem outros não tão fortes que são utilizados como piada ou somente para incomodar, se espalhar pelos computadores sem fazer mal à máquina e sim à paciência do usuário.

### 2.2 História

O primeiro antivírus foi criado em 1988 por Denny Yanuar Ramdhani. Era uma vacina ao vírus Brain, um vírus de boot, além de remover o vírus imunizava o sistema contra uma nova infecção. A forma de desinfetar era remover as entradas do vírus no pc e já bloqueava estas fraquezas para impedir um novo ataque. Ainda em 1988 um vírus foi projetado para infectar com a "ajuda" da BBS, neste John McAfee, desenvolveu o VirusScan, primeira vacina para o vírus.

### 2.3 Antivírus e SO

Por enquanto existe uma dependência dos vírus para com os sistemas operacionais, pois afetam o modo em que o executável interage com o sistema, e pedidos

especiais são feitos pelo próprio SO e cada qual o faz de forma diferente, ou seja um vírus que funciona em windows nunca funcionaria em linux, só se fossem chamadas suas APIs, como feito pelo wine no sistema linux, e mesmo assim não teria todo o potencial de infecção, já que é preparado para a estrutura do sistema para o qual foi projetado.

### **2.3.1 DOS**

No sistema DOS o anti-virus não funciona em "tempo real", somente como scanner, normalmente era colocado no boot do sistema para varrer o sistema em busca de novas infecções, e outras verificações somente se chamado pelo usuário. Sendo infectado no meio de uma tarefa o vírus já se propagou e danificou diversas áreas e somente será percebido na nova execução do antivírus.

### **2.3.2 Windows**

Já no windows o antivírus protege as principais formas de ataque, para este sistema. continua a utilizar o scanner, como no DOS. Ganhou a função de monitoramento, com diversas ferramentas para encontrar padrões de vírus. A cada executável aberto há esta verificação, o que compromete o desempenho do computador. A cada período pré-determinado há uma varredura sobre os arquivos do sistema para verificar arquivos infectados, remove o vírus e tenta manter a integridade do arquivo. Se encontra um padrão de infecção mas ainda não existe "vacina" para remoção diversos sistemas de proteção utilizam a ferramenta de "quarentena", ou seja mantém o arquivo infectado em um espaço que não pode ser "alcançado" pelo usuário até que possa restaurar o arquivo, ou ao menos conheça o vírus.

### **2.3.3 Linux**

Não são muito populares neste sistema. Por enquanto não há uma grande preocupação, nem pela parte de usuários e nem pela parte de desenvolvedores. O que existe hoje são alguns sistemas que detectam vírus para windows pelo linux, para fazer uma manutenção do sistema. E mesmo assim não são tão "potentes" quanto os de windows, não há muita preocupação em desenvolvê-los.

## 2.4 Técnicas de detecção

São diversas as técnicas de detecção dentre elas: Heurística: Que significa descobrir. Estuda o comportamento, estrutura e características para analisar se é perigoso ao sistema ou inofensivo. Emulação: Abre o arquivo em uma virtualização do sistema, e analisa os efeitos sobre o sistema. Arquivo monitorado: Mantém um arquivo no sistema e o monitora, se ele modificar alguma característica é porque o sistema foi infectado. E então o antivírus toma as precauções necessárias. Assinatura do vírus: Com um trecho de código do vírus tem-se sua assinatura, quando tenta detectar o vírus busca-as para analisar se já não existe dentro do banco de dados do antivírus. Temos o falso positivo, o antivírus com base no comportamento do arquivo o considera infectado, o que dificulta para usuários comuns identificarem as anomalias e utilizar com segurança o sistema.

### 2.4.1 Virus de pendrive

No sistema operacional windows eles se utilizam do arquivo autorun.inf para se autoexecutar e infectar a máquina. sua limpeza é simples, existem alguns antivírus que alteram o conteúdo do autorun e tiram a permissão de gravação do arquivo, e alguns usuários criam um diretório com o nome autorun.inf e isso impede de criar o tal arquivo. os vírus em si funcionam de forma interessante, temos por exemplo o conficker q após infectar o pc ele passa a infectar td pendrive q nel for utilizado, assim como enquanto conectado a internet ele baixa diversos outros vírus e com isso acaba com o sistema e arquivos do usuário. sua prevenção é simples e sua remoção é complicada. ou seja se todos fossem informados de como o vírus funciona a prevenção seria óbvia e este tipo de vírus seria obsoleto.

### 2.4.2 Virus de macro

Os vírus de macro são utilizados dentro de, aparentemente, inofensíveis arquivos estilo "office" são scripts executados automaticamente para facilitar a visualização dos arquivos e fazer eles executarem o que teriam de executar, os criadores de vírus aproveitam que macros tem poder de execução e infectam os arquivos colocando dentro a macro código malicioso que o usuário previamente nem notará, e após execução do arquivo já estará infectado e infectará outros. A maior praga disso está nas apre-

sentações de slides, como foi muito difundido por e-mails para passar imagens com animações. O vírus se instala dentro destes arquivos e o usuário desconhece que por trás de tudo que está visualizando um vírus acabou de se instalar em sua máquina.

### **2.4.3 Virus Polimórficos**

Ainda não existe uma forma eficaz para se detectar este tipo de vírus, eles não tem um padrão a ser identificado. O que se faz é criar um arquivo de vítima e este fica sempre sendo monitorado, mas o bom vírus polimórfico já está residente em memória e faz o sistema "ver" o arquivo como inalterado e com isso não há mais nada a ser feito. seria uma limpeza manual, sem o auxílio de outra máquina seria inviável, enquanto o vírus se infecta o usuário tentaria localizá-lo e deletá-lo uma guerra perdida.

## Polimorfismo

### 3.1 O que é o polimorfismo?

Segundo o dicionário Aurélio<sup>1</sup>

**Polimorfismo.** s.m. Qualidade do que é polimorfo.

**Polimorfo.** adj. Que é sujeito a mudar de forma. Que se apresenta sob diversas formas.

O polimorfismo está diretamente ligado à criptografia. O principal objetivo de um código polimórfico é que duas versões não tenham nada em comum visualmente, evitando assim que o conteúdo seja facilmente identificável e associado à sua verdadeira função.

#### 3.1.1 Polimorfismo usado de forma legal

Quando se ouve falar em código polimórfico, a primeira associação que se faz é com vírus de computador. No entanto, existem diversas aplicações legais do código polimórfico.

Empresas que desenvolvem código de proteção contra cópia ilegal de software usam código polimórfico para dificultar a engenharia reversa do software de proteção e do software protegido visando dificultar o trabalho de crackers que criam patches para fazer com que o software pense que está registrado legalmente<sup>2</sup>. Atualmente, todas as empresas de proteção contra cópia usam polimorfismo juntamente com compactação

<sup>1</sup><http://www.dicionariodoaurelio.com/>

<sup>2</sup>Exemplos: <http://www.webtoolmaster.com/packer.htm> e <http://www.pelock.com/products/pelock>

de dados e técnicas anti-debugging para proteger o software. Alguns usam também códigos metamórficos. O código metamórfico difere do polimórfico por usar técnicas de recompilação/reconstrução fazendo com que cada versão seja única não somente na aparência mas também no código binário executável.

Outro uso legal do código polimórfico seria para gerar uma marca digital do software, pois cada versão seria única tornando possível assim dizer quem é o dono da versão que está em circulação no caso de haver alguma cópia ilegal. A indústria fonográfica já usa algo semelhante a isso nas músicas<sup>3</sup> com a finalidade de descobrir quem disponibilizou cópias ilegais. O mais importante é que esta medida não altera em nada a qualidade do áudio, passando totalmente despercebida pelo usuário. Também é usado este artifício em máquinas fotográficas e dispositivos de gravação. Cada dispositivo possui sua marca digital única fazendo assim sua 'assinatura' em cada foto ou filme produzidos pelo dispositivo.

### 3.1.2 Como funciona?

Um código polimórfico exige no mínimo duas partes: a rotina de encriptação e a rotina de decriptação. A criptografia pode ser algo super simples ou algo muito complexo.

Exemplo:

A função lógica  $\oplus$  (ou exclusivo) faz a combinação binária

A	B	$C = A \oplus B$	$C \oplus B$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Neste exemplo, A representa o valor de 8, 16, 32 ou 64 bits a ser codificado e B representa a chave de codificação. C é o resultado após a codificação e a última coluna mostra que se aplicarmos o valor codificado à chave original, obtemos novamente o valor original. A função  $\oplus$  é largamente utilizada devido a esta propriedade de facilmente restaurar o valor original.

Apesar desta facilidade, um código feito com esta codificação simplista pode ser facilmente detectado através de análise de padrões. A seguir, vamos demonstrar o uso

---

<sup>3</sup>[http://en.wikipedia.org/wiki/Digital\\_watermarking](http://en.wikipedia.org/wiki/Digital_watermarking)

da codificação via operação ou exclusivo, usando o código abaixo.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void) {
    char texto[255];
    char codificado[255];
    int i, chave;

    printf("Digite o texto a ser codificado (máx. 255 caracteres)\n");
    gets(texto);
    printf("Digite a chave [1..255] para codificar o texto: ");
    scanf("%d", &chave);
    printf("Codificando...\n");
    for (i=0; i < strlen(texto); i++)
        codificado[i] = texto[i] ^ (char) chave;
    printf("Texto codificado: [%s]\n", codificado);
    printf("Decodificando...\n");
    for (i=0; i < strlen(codificado); i++)
        codificado[i] = codificado[i] ^ (char) chave;
    printf("Texto decodificado: [%s]\n", codificado);
    return 0;
}
```

Vamos usar este código para fazer alguns exemplos

Texto original	Chave	Texto codificado
A vida passa depressa!	1	@!whe'!q'rr'!edqsdr'
A vida passa depressa!	23	V7a sv7gvddv7srgerddv6
A vida passa depressa!	24	Y8nq y8hykky8 }hj}kky9
ABCDEFGHIJKLM	13	LONIHKJEDGFA
abcdefghijkl	13	lonihkjedgfa
abc,abc,abc	31	lon!lon!lon
aaaaaaaaaa	54	WWWWWWWWWWW

Ao olharmos para as três primeiras linhas, a codificação parece ser promissora e parece ser bem difícil, sem ter conhecimento do texto original, e nem da chave, adivinhar o que está escrito no texto codificado. Mas basta um olhar mais atento às 4 últimas linhas e veremos claramente o problema: a formação de padrões.

Então, se há formação de padrões, a codificação torna-se inútil pois um analisador de padrões, ou mesmo uma pessoa, através de análise dos dados e dos padrões de repetição da língua poderiam facilmente encontrar o texto original. No nosso exemplo



usamos um texto mas o mesmo princípio aplica-se ao código binário executável e às instruções do processador. Existe um número finito de instruções e suas combinações são bastante conhecidas e portanto descobrir a codificação, ainda que seja uma tarefa trabalhosa, é perfeitamente possível.

Para evitar a formação de padrões, existem várias técnicas que podem ser aplicadas:

- Rotacionar bits da palavra codificada. Uma sequência de bits tem um bit mais significativo e um menos significativo. Um byte, por exemplo, pode ser descrito como 76543210, onde 7 é a posição do bit mais significativo e 0 é a posição do bit menos significativo. A rotação de bits pode ser simples: 07654321. Neste caso, o byte foi rotacionado um bit para a direita, ficando o bit menos significativo na posição do mais significativo. Pode parecer uma coisa bem simples, mas usado em combinação com a operação ou exclusivo, esta é uma técnica que dificulta bastante a decodificação. Principalmente porque pode ter rotação variável. Exemplo: os bits podem rotacionar de acordo com sua posição, ou seja os bytes de posição par podem rotacionar 2 bits e os da posição ímpar um bit. Ou ainda: Múltiplos de 3, não rotaciona; ímpares rotacionam um para a esquerda e pares 2 rotacionam 1 para a direita. Enfim, o limite da combinação é a imaginação do criador.
- Troca de posição dos bytes. Embora mais trabalhosa, é uma técnica interessante. Imaginemos um palavra de 32 bits, onde cada byte é representado pelas letras ABCD, sendo que o byte mais significativo o A e o menos significativo o D. É possível apenas rotacionando os bits formar BADC ou ainda ACDB, ou qualquer outra combinação desejada. Para dificultar mais a vida de quem vai analisar, pode-se fazer isso de forma não convencional, por exemplo usando grupos de 3 bytes, em vez de 4.
- Somar a palavra atual com a anterior já codificada, ou aplicar a operação ou exclusivo com a palavra anterior. Neste caso a codificação acontece do início para o fim e a decodificação acontece do fim para o começo. No caso da soma ou subtração tem que tomar cuidado extra por causa do overflow ou underflow. Por exemplo, se somar 20 ao byte de valor 250 o resultado seria 270, que não é possível de representar em 8 bits. No caso da soma a técnica usada é trabalhar usando módulo. No caso,  $270 \% 256 = 14$ . Na verdade o overflow/underflow são desejados pois ajudam a camuflar os resultados.

- Embaralhar a ordem da informação. Exemplo: depois de codificado, trocar os itens de posição par com os de posição ímpar.
- Inserção de lixo no meio dos dados reais. Exemplo: a cada 50 bytes codificados, é inserido 7 bytes de lixo. Este lixo obviamente não tem significado algum e será ignorado na decodificação. No entanto, atrapalha em muito quem está tentando decifrar o código, uma vez que não tem como saber se os dados fazem ou não parte do código real.
- Utilização de chaves de tamanho variado. A chave pode ser uma sequência de números aleatórios, de tamanho aleatório. Esta sequência aleatória geralmente não é aleatória mas sim pseudoaleatória<sup>4</sup>. Ou é usada uma tabela pré-definida ou uma função que gera a sequência à partir da chave inicial. Neste caso, a grande dificuldade é descobrir a tabela ou a função que gera a chave.

Vamos modificar a nossa rotina para usar a operação ou exclusivo juntamente com rotação de bits para vermos se conseguimos nos livrar da formação de padrões. O código modificado está listado abaixo.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

unsigned char rol(unsigned char c, int bits)
{
    return ((c << bits) & 255) | (c >> (8 - bits));
}

unsigned char ror(unsigned char c, int bits)
{
    return (c >> bits) | ((c << (8 - bits)) & 255);
}

void imprime_hex(char str[], int tam) {
    int i;

    for (i=0; i < tam; i++)
        printf("%02X", (unsigned char) str[i]);
    printf("\n");
}

int main(void) {
    char texto[255];
```

---

<sup>4</sup>[http://pt.wikipedia.org/w/index.php?title=Sequ%C3%Aancia\\_pseudoaleat%C3%B3ria&oldid=30480084](http://pt.wikipedia.org/w/index.php?title=Sequ%C3%Aancia_pseudoaleat%C3%B3ria&oldid=30480084)

```
char  codificado[255];
int   i, c, shift, chave;

printf("Digite o texto a ser codificado: ");
gets(texto);
printf("Digite a chave [1..255] para codificar o texto: ");
scanf("%d", &chave);
printf("Codificando...\n");
shift = 3;
/* codificação */
for (i=0; i < strlen(texto); i++) {
    /* aplica ou exclusivo */
    c = (unsigned char) texto[i] ^ chave;
    /* rotaciona os bits */
    if (i % 2 == 0)
        c = ror(c, shift);
    else
        c = rol(c, shift);
    /* guarda o byte codificado */
    codificado[i] = (char) c;
    shift = (++shift % 3) + 1;
}
printf("Devido ao fato da codificação gerar caracteres não ASCII,\n");
printf("o texto codificado será apresentado em hexadecimal:\n");
imprime_hex(codificado, strlen(texto));
printf("Decodificando...\n");
shift = 3;
/* decodificação */
for (i=0; i < strlen(texto); i++) {
    c = codificado[i];
    /* rotaciona os bits */
    if (i % 2 == 0)
        c = rol(c, shift);
    else
        c = ror(c, shift);
    /* aplica ou exclusivo */
    codificado[i] = (char) c ^ chave;
    shift = (++shift % 3) + 1;
}
printf("Texto decodificado: [%s]\n", codificado);
return 0;
}
```

O código mudou e agora a saída não pode mais ser em texto por conta de que certamente teremos muitos caracteres não ASCII. Optamos por mostrar o resultado em hexadecimal. A mudança em relação ao primeiro código que implementamos está

apenas na utilização de rotação de bits. Fizemos um código simples apenas para ser didático e mostrar o funcionamento na prática. Usamos um artifício de que quando a posição que estamos codificando for par fazemos a rotação à esquerda. Quando for ímpar, fazemos a rotação à direita. Também variamos o número de bits rotacionados de 1 a 3, conforme vamos avançando na codificação. Novamente frisamos que trata-se de um código simples, sem pretensão de ser perfeito nem otimizado. Codificações de criptografia forte com análise estatística de padrões podem ser encontradas em livros que tratam do assunto de criptografia.

Vamos repetir novamente o estudo da tabela anterior, usando este novo código.

Texto original	Chave	Texto codificado
A vida passa depressa!	1	08 84 BB 43 59 C0 24 C5 30 93 9C C0 24 95 32 8B DC C8 4E C9 30 01
A vida passa depressa!	23	CA DC B0 F3 DC EC E6 9D 3B 23 19 EC E6 CD 39 3B 59 E4 8C 91 3B B1
A vida passa depressa!	24	2B E0 37 8B 1F F2 07 A1 BC 5B DA F2 07 F1 BE 43 9A FA 6D AD BC C9
ABCDEFGHIJKLM	13	89 3D 27 4A 12 96 49 15 22 3A 91 82
abcdefghijkl	13	8D BD 37 4B 1A D6 4D 95 32 3B 99 C2
abc,abc,abc	31	CF F5 3E 99 9F FA 8F CC 3F EB 1F
aaaaaaaaaa	54	EA 5D AB BA D5 AE EA 5D AB BA

Note que apesar de melhorar muito, na última linha ainda conseguimos ver perfeitamente um padrão. No entanto, quem olha para o padrão não poderá facilmente imaginar que este padrão é um único caractere. Para ficar mais claro o padrão, codificamos novamente a última linha com 20 caracteres e obtivemos:

**EA 5D AB BA D5 AE EA 5D AB BA D5 AE EA 5D AB BA D5 AE EA 5D**

Pode-se ver claramente o padrão agora que separamos a parte de uma cadeia mais longa. Entretanto, este padrão pode ser diminuído e talvez até eliminado somando-se ao caractere a posição dele na linha em módulo 255 e fazendo a operação ou exclusivo com o caractere codificado anteriormente. O padrão ainda poderá repetir-se quando a codificação resultar em 00. No entanto, já geramos muitos dígitos diferentes para dificultar bastante a decodificação. Não esquecendo que este padrão só é perceptível por conta de que estamos codificando sempre o mesmo caractere N vezes, o que não ocorre em uma situação real.

A título de curiosidade, o resultado após implementar a soma posicional e a op-

eração ou exclusivo com o caracter anterior pode ser visto abaixo. Note que o único caracter que continua igual é o primeiro pois ele não tem nenhum anterior para aplicar a operação e a posição dele é 0.

EA B4 19 A4 7D CE 3E 5A E9 2A F5 4C BA D0 69 A0 45 FA 06 76

Caso não tenha entendido o que aconteceu, o segundo caractere gerado foi **5D** na codificação anterior. Na nova codificação, foi aplicado  $5D + 1 = 5E$  em seguida  $5E \oplus EA$  o que resultou em **B4**. O próximo caracter gerado era **AB**. A este caracter foi aplicado  $AB + 2 = AD$  em seguida  $AD \oplus B4$ , o que resultou em **19** e assim sucessivamente. Note que esta codificação é muito mais poderosa pois é feita baseada no caracter anterior e na posição física do caracter. Além disso, acrescentamos uma dificuldade maior, imposta pelo próprio algoritmo, que é a decodificação do final para o começo.

Agora que entendemos como é feita a criptografia, entender o código polimórfico torna-se muito simples. Conforme dissemos anteriormente, é necessário uma rotina de codificação e outra de decodificação. Sem tais rotinas não temos como desenvolver o processo. Em geral a rotina de codificação está em outro módulo - no caso dos software de proteção anti-cópia - ou é codificada junto como restante do código a ser protegido - no caso dos vírus de computador.

Logo de início a primeira pergunta que vem à mente é: **Mas se existe uma rotina de decodificação, de que adianta o resto estar codificado? Não basta apenas executar a rotina para obter o código original novamente?** De fato, de posse da rotina de faz a decodificação do código criptografado seria muito simples obter o código original. No entanto, existem várias técnicas para impedir o acesso à rotina de decodificação. Este é um dos assuntos do próximo capítulo.

## O vírus polimórfico

Na época do MS-DOS os vírus eram simples e divididos em categorias básicas: BOOT-SECTOR, infectantes de arquivos .COM e infectantes de arquivos .EXE. Nesta época a vida também era relativamente fácil para os fabricantes de anti-vírus pois os vírus eram em menor número e a detecção era baseada em assinatura do código malicioso<sup>1</sup>. As atualizações dos anti-vírus eram em geral atualização da base de dados que continham as assinaturas, o tamanho e a forma de correção da infecção.

Um exemplo desta época é o vírus de boot-sector **Stoned**<sup>2</sup> que infectou muitos computadores no final da década de 1980. A assinatura mais óbvia para este vírus seria *Your PC is now Stoned!* que o vírus exibia quando o computador estava inicializando. Portanto, um anti-virus da época precisaria apenas buscar esta string no registro de boot sector do disco rígido e dos disquetes que estivessem na unidade e, caso encontrasse, eliminar o vírus da memória - pois ele ficava residente infectando todo disquete que colocasse no computador - e em seguida substituir o boot sector pelo original que o vírus mantinha em outra localização do disco.

Algumas versões de vírus de boot sector eram um pouco mais inteligentes e assumiam controle da função de leitura de disco do BIOS. Assim, ao detectar que algum software estava tentando ler o boot sector, ele carregava a cópia original fazendo com que o anti-vírus não suspeitasse da existência da infecção. Logo, os desenvolvedores de anti-vírus perceberam esta manobra e começaram a vasculhar a memória RAM do computador em busca de assinaturas de vírus e não mais somente em disco.

Também começaram a surgir cada vez mais vírus e a detecção por assinatura somente não estava mais dando certo pois novas variações do mesmo vírus tinham

<sup>1</sup>A assinatura de um vírus é um padrão de bytes que identifica unicamente aquele vírus

<sup>2</sup>[http://en.wikipedia.org/w/index.php?title=Stoned\\_%28computer\\_virus%29&oldid=532807447](http://en.wikipedia.org/w/index.php?title=Stoned_%28computer_virus%29&oldid=532807447)

assinaturas diferentes. Por exemplo, o Stoned mencionado anteriormente teve muitas variações e buscar pela assinatura original não detectava mais o vírus pois a mensagem foi modificada. Então as empresas de anti-vírus começaram a desenvolver algoritmos que analisavam o código a fim de detectar certos padrões de execução (chamado código malicioso) que identificavam por certo um código que não deveria ser executado.

Então, os desenvolvedores de vírus perceberam que para evitar a detecção deveriam modificar a aparência do código. Foi assim que surgiram os vírus polimórficos.

# Capítulo 5

## Conclusão



# Referências Bibliográficas

- [1] Mental Driller. Interview with the mental driller/29a, março de 2002. Disponível em <http://www.thehackademy.net/madchat/vxdevl/interviews/mentaldriller%2301.txt>.
- [2] Mark A. Ludwig. *The Giant Black Book of Computer Viruses*. American Eagle Publications, Inc, Post Office Box 1507. Show Low, Arizona, 1 edition, 1995.
- [3] Peter Szor. Hunting for metamorphic. Disponível em <http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf>.
- [4] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, fevereiro de 2005.
- [5] Wikipedia. Computer virus — Wikipedia, The Free Encyclopedia, 2012. [Online; acessado em 11-Abr-2012].. Disponível em [http://en.wikipedia.org/w/index.php?title=Computer\\_virus&oldid=486444876](http://en.wikipedia.org/w/index.php?title=Computer_virus&oldid=486444876).

## **Apêndices**

# Estrutura de Arquivos PE e ELF

## A.1 Arquivo PE

O formato de arquivo PE (Portable Executable Format File) é o último utilizado para plataforma Microsoft.

### A.1.1 Estrutura de arquivo PE.

DOS 2 - Cabeçalho EXE compatível	Seção DOS 2.0 (para compatibilidade com DOS somente)
Não utilizado	
OEM - Identificador	
OEM - Info	
Offset para cabeçalho PE	
DOS 2.0 Stub Program & Reloc. Table	
Não utilizado	
PE - Cabeçalho	Palavras limitadas a 8 bytes
Tabela de seções	
Image Pages	
· Info de Importação	
· Info de Exportação	
· Info de correção	
· Info de recursos	
· Info de debug	

### A.1.2 PE - Cabeçalho

Temos no cabeçalho uma estrutura dividida em campos com palavras de 4 bytes, enfatizamos alguns deles abaixo:

Tipo de CPU: o campo informa qual o tipo de CPU para a qual o executável foi projetado.

Número de Seções: o campo informa o número de entradas na tabela de seções.

Marca de Tempo/Data: Armazena a data de criação ou modificação do arquivo.

Flags: Bits para informar qual o tipo de arquivo ou quando há erros em sua estrutura.

LMAJOR/LMINOR: maior e menor versão do linkador para o executável.

Seção de alinhamento: O valor de alinhamento das seções. Deve ser múltiplo de 2 dentre 512 e 256M. O valor padrão é 64K.

OS MAJOR/MINOR = Versões limitantes (maior e menor) do sistema operacional.

Tamanho da Imagem: Tamanho virtual da imagem, contando todos os cabeçalhos. E o tamanho total deve ser múltiplo da seção de alinhamento.

Tamanho do Cabeçalho: Tamanho total do cabeçalho. O tamanho combinado de cabeçalho do DOS, cabeçalho do PE e a tabela de seções.

FILE CHECKSUM: Checksum do arquivo em si, é setado como 0 pelo linkador.

Flags de DLL: Indica qual o tipo de leitura que deve ser feita, processos de inicialização e terminação de leitura e de threads.

Tamanho reservado da pilha: tamanho de pilha reservado ao programa, o valor real é o valor efetivo, se o valor reservado não tiver no sistema ele será paginado.

Tamanho efetivo da pilha: tamanho efetivo.

Tamanho Reservado da HEAP: Tamanho reservado a HEAP.

Tamanho efetivo da HEAP: Valor efetivo para a HEAP.

### A.1.3 Tabela de Seções

O número de entradas da tabela de seções é dado pelo campo de número de seções que está no cabeçalho. As entradas se iniciam em 1. Segue imediatamente

o cabeçalho do PE. A ordem de dados e memória é selecionado pelo ligador. Os endereços virtuais para as seções são confirmados pelo ligador de forma crescente e adjacente, e devem ser múltiplos da Seção de alinhamento, que também é fornecida no cabeçalho do PE. Abaixo alguns de uma seção nesta tabela, divididos em palavras de 8 bytes:

Nome da Seção: Campo com 8 bytes nulos para representar o nome da seção em ASCII.

Tamanho virtual: O tamanho virtual é o alocado quando a seção é lida.

Tamanho físico: O tamanho de dados inicializado no arquivo para a seção. É múltiplo do campo de alinhamento do arquivo do cabeçalho do PE e deve ser menor ou igual ao tamanho virtual.

Offset físico: Offset para apontar a primeira página da seção. É relativo ao início do arquivo executável.

Flags da seção: Flags para sinalizar se a seção é de código, se está inicializada ou não, se deve ser armazenada, compartilhada, paginável, de leitura ou para escrita.

#### **A.1.4 Páginas de imagem**

A página de imagens contém todos os dados inicializados e todas as seções. As seções são ordenadas pelo endereço virtual reservado a elas. O Offset que aponta para a primeira página é especificado na tabela de seções como visto na subseção acima. Cada seção inicia com um múltiplo da seção de alinhamento.

#### **A.1.5 Importação**

A informação de importação inicia com uma tabela de diretórios de importação que descreve a parte principal da informação de importação. A tabela de diretórios de importação contém informação de endereços que são utilizados nas referências de correção para pontos de entrada com uma DLL. A tabela de diretórios de importação consiste de um vetor de entradas de diretórios, uma entrada para cada referência a DLL. A última entrada é nula o que indica o fim da tabela de diretórios.

### A.1.6 Exportação

A informação de exportação inicia com a tabela de diretórios de exportação que descreve a parte principal da informação de exportação. A tabela de diretórios de exportação contém informação de endereços que são utilizados nas referências de correção para os pontos de entrada desta imagem.

### A.1.7 Correção

A tabela de correção contém todas as entradas de correção da imagem. O tamanho total de dados de correção no cabeçalho é o número de bytes na tabela de correção. A tabela de correção é dividida em blocos de correção. Cada bloco representa as correções para um página de 4K bytes. Correções que são resolvidados pelo ligador necessitam ser processadas pelo carregador, a menos que a imagem não possa ser carregada na Base de imagens especificada no cabeçalho do PE.

### A.1.8 Recursos

Recursos são indexados por uma árvore binária ordenada. O design como um todo pode chegar a  $2^{31}$  níveis, entretanto, NT utiliza somente 3 níveis: o mais alto com o *tipo*, no subsequente *nome*, depois a *língua*.

### A.1.9 Debug

A informação de debug é definido por um debugador que não é controlado pelo PE ou pelo ligador. Somente é definido pelo PE os dados da tabela de diretório de debug.

## A.2 Arquivo ELF

Explicação

### A.2.1 A estrutura do arquivo ELF

Arquivo Realocável	Arquivo Carregável
Cabeçalho ELF	Cabeçalho ELF
Tabela do cabeçalho do programa (opcional)	Tabela do cabeçalho do programa
seção 1	Segmento 1
seção 2	
...	Segmento 2
seção n	...
Tabela de cabeçalho de seção	Tabela de cabeçalho de seção(opcional)