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
- Ao professor Bruno por toda a paciência que teve conosco.
- A todas as pessoas que direta ou indiretamente contribuiram para a nossa formação.

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

Capítulo 1

Introdução

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

Uma pesquisa feita pela pela Symantec[**?** **?**] em 24 países com 13 mil usuários entre 18 e 64 anos, aponta que em 2012 o crime virtual gerou um prejuízo global de mais de 100 bilhões de dólares! Este levantamento, feito anualmente pela Symantec, mostra que existe um grande número de vítimas deste tipo de crime: são aproximadamente 18 vítimas por segundo, ou 556 milhões de vítimas por ano! Ainda segundo o estudo, dois terços da população que usa a internet já foi vítima de algum tipo de crime virtual.

O Brasil aparece em quarto lugar no ranking de países com maior atividade de crimes na internet[**?**], e está entre os que tem maior prejuízo com este tipo de crime. Estima-se que 28 milhões de pessoas foram vítimas em 2012, causando um prejuízo total de 15 bilhões de reais, ou aproximadamente 562 reais por vítima.

Um dos principais fatores que ajudam no aumento destes casos é o acesso cada vez maior da população à internet através de computadores pessoais e, principalmente, smartphones e tablets. As redes sociais, como o facebook, tornaram-se populares entre estes usuários que ainda estão dando os primeiros passos na vida virtual, o que os torna as principais vítimas justamente por não terem usado esta tecnologia anteriormente, nem estarem familiarizados com virus, worms, phishing, etc. Segundo o estudo, dois terços dos usuários não utilizam nenhum sistema de segurança (antivi-

rus, por exemplo) para dispositivos móveis e 44% destes usuários nem sequer sabe que tais sistemas existem.

Ainda,segundo o estudo:

- 40% dos usuários não sabem que um vírus podem passar despercebidos, sem causar nenhum dano aparente ao sistema, o que torna praticamente impossível ao usuário notar que existe o virus na sua máquina.
- 49% concordam que sem o computador parar ou ficar muito lento, não teriam como saber se tem um virus ou malware instalado em seu computador.
- 55% não tem certeza se seu computador está livre de vírus.
- 3/10 não entendem os riscos virtuais (cybercrimes) e nem sabem como se proteger.
- 30% não se preocupam com cybercrime quando estão online, simplesmente porque não acreditam que possam ser vítimas.
- 21% não protegem suas informações pessoais quando estão online.

Mas ainda temos alguma esperança:

- 89% apagam email suspeito de pessoas que não conhecem
- 83% possuem ao menos um anti-virus básico
- 78% não abrem links ou anexos de e-mails ou textos que eles não solicitaram.

O conhecimento é a chave para esclarecer e desmistificar. Esta foi a principal motivação para escrever este trabalho.

Capítulo 2

Revisão bibliográfica

2.1 Antivírus

Os antivirus [**?**] são softwares criados para analisar, detectar, eliminar e impedir os virus informáticos ou ao menos diminuir a intensidade do ataque. Foram criados pela necessidade de que os virus impediam a utilização do sistema. Os virus 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 antivirus foi criado em 1988 por Denny Yanuar Ramdhani. Era uma vacina ao virus Brain, um virus de boot, além de remover o virus imunizava o sistema contra uma nova infecção. A forma de desinfectar era remover as entradas do virus no pc e já bloqueava ests fraquezas para impedir um novo ataque. Ainda em 1988 um virus foi projetado para infectar com a "ajuda"da BBS, nisto John McAfee, desenvolveu o VirusScan, primeira vacina para o virus.

Abaixo segue um resumo da história da evolução dos vírus:

Em 1982 um programador de 15 anos (Rich Skrenta), criou o primeiro código malicioso, para Apple 2 em DOS, não fazia mal algum, mas a um certo número de execuções ele exibia uma poesia criado pelo autor.

Ainda em 1983, houve o pesquisador Fred Cohen, que deu o nome a programas

com códigos nocivos aos computador de “Vírus de computador”. Neste ano também Len Eidelmen demonstrou um um seminário um programa que se replicava em vários locais do sistema. Em 1984 na 7th Annual Information Security Conference, o termo vírus passou para programa que infecta outros programas e os modifica para produzir novas cópias de si mesmo.

Já em 1986 teve sim o primeiro vírus considerado, chamado Brain, que era um vírus de boot, danificava o setor de inicialização do disco rígido, se propagava através de disquetes contaminados. Foi elaborado pelos irmãos Basit e Amjad. A falha utilizada é que os endereços da memória eram físicos, então alterar o bloco que inicia o boot era simples. No ano seguinte foi criado o vírus Vienna, a cada execução ele infectava arquivos com extensão COM. Aumentavam o tamanho do executável em 684 bytes, os programas não tinham uma cópia dele só era alterados, foi criado o ThusFix que neutralizava o Vienna, não foi considerado um antivírus e sim somente uma correção.

1988, o primeiro antivírus foi criado, contra o Brain, este escrito por Denny Yanuar Ramdhanj. Removia as entradas do vírus e imunizava contra novos ataques. No ano seguinte foi criado o Dark Avenger, contaminava rapidamente as máquinas e as destruía lentamente, para que ele não fosse percebido até que fosse muito tarde, então a IBM criou o primeiro antivírus com finalidade comercial, no inicio do ano haviam 9% das máquinas contaminadas e ao final do ano 63%. Em novembro Robert Morris Jr, que era um especialista da Agencia Nacional de Segurança no EEUU, contruiu um vírus chamado Morris que utilizava uma falha do Unix e atacou 6 mil servidores, por volta de 10% das máquinas com internet da época, foi condenado a serviço comunitário e uma multa de 10 mil dólares. Em dezembro de 90 iniciou a entrada de mais empresas no ramo de antivírus, eram elas: Symantec, McAfee, IBM e Kaspersky.

Somente em 1992 que o vírus apareceu a mídia, com o nome de Michelangelo, sobrepunha partes do disco e criava diretórios e arquivos falsos no dia 6 de março, que era o dia do nascimento da renascença. Por este motivo a venda de antivírus cresceu muito.

1995, pela primeira vez um criador de vírus é preso, foi rastreado pela Scotland Yard. Cristopher Pile, conhecido também como Barão Negro, condenado a 18 meses de prisão. Seu vírus era o Pathogen, o estrago dele foi em quase 2 milhões de dólares e meio, Barão Negro era um programador desempregado do sudeste da Inglaterra. Foi encontrado, e então confessou as 5 acusações de invasão a computadores para diminuir sua condenação. 5 modificações não autorizadas de software e uma acusação de incitação a propagar seu vírus. Neste ano também era criado o Concept, primeiro

do tipo de vírus de macro, escrito para o Word em Basic, poderia ser executado em qualquer plataforma com Word ou Macintosh, se espalhava pelo setor de boot e por todos os executáveis.

1999, em taiwan o CIH infectava até o windows ME, era residente em memória e podia sobrescrever dados no HD, o que o tornava inoperante, ficou conhecido também como Chernobyl,ficou inofensivo quando houve a migração para o sistema NT aos usuários Windows residenciais, foram estimados cerca de 291 milhões de dolares em prejuízo.

Em 2000 o vírus Love Letter foi solto, originado nas Filipinas, varreu a Europa e os Estados Unidos em cerca de 6 horas, infectou cerca de 3 milhões de máquinas e o dano de quase 9 milhões de dólares. No ano seguinte a “moda” são os worms, propagados por e-mail, redes sociais e muitas outras páginas da internet.

Em 2002 David L. Smith, foi sentenciado a 20 meses de cadeia pelos danos causados pelo seu vírus, batizado de Melissa. Causou milhões de dólares de danos. Em 1999 ele já se declarou culpado da acusação de roubo de identidade de um usuário de internet para enviar o seu “programa”, seria condenado a 5 anos de prisão, mas por auxiliar a encontrar outros autores de vírus, baixou sua pena para 2 anos e uma multa de 5 mil dólares.

2007 houve o aumento dos vírus em redes sociais com chamadas que atraiam a vitima a clicar e acabaram se infectando e enviando mensagens contaminadas a todos os que se “uniam” a ele naquela rede. O Arquivo era um arquivo pequeno que envia mensagens a todos os contatos e também pode roubar senhas de banco.

2.3 Antivirus e SO

Por enquanto existe uma dependência dos virus 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 virus 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 virus já se propagou e danificou diversas areas e somente será percebido na nova execução do antivirus.

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 virus. A cada executável aberto há esta verificação, o que compromete o desempenho do computador. A cada periodo pré-determinado há uma varredura sobre os arquivos do sistema para verificar arquivos infectados, remove o virus 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 virus.

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 virus 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 desenvolve-los.

2.4 Técnicas de detecção

São diversas as técnicas de detecção dentre elas: Heuristica: Que significa descobrir. Estuda o comportamento, estrutura e caracteristicas 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 caracteristica é porque o sistema foi infectado. E então o antivirus toma as precauções necessárias. Assinatura do virus: Com um trecho de código do virus tem-se sua assinatura, quando tenta detectar o virus busca-as para analisar se já não existe dentro do banco de dados do antivirus. Temos o falso positivo, o antivirus com base no comportamente 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 antivirus que alteram o conteudo do autorun e tiram a permissão de gravação do arquivo, e alguns usuários criam um diretorio com onome autorun.inf e isso impede de criar o tal arquivo. os virus em si funcionam de forma interessante, temos por exemplo o conficker q apos infectar o pc ele passa a infectar td pendrive q nel for utilizado, assim como enquanto conectado a internet ele baixa diversos outros virus e com isso acaba com o sistema e arquivos do usuario. sua prevenção é simples e sua remoção é complicada. ou seja se todos fossem informados de como o virus funciona a prevenção seria óbvia e este tipo de virus seria obsoleto.

2.4.2 Virus de macro

Os virus de macro são utilizados dentro de, aparentemente, inofensiveis 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 virus aproveitam que macros tem poder de execução e infectam os arquivos colocando dentre 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 esta nas apresentações de slides, como foi muito difundido por e-mails para passar imagens com animações. O virus se instala dentro destes arquivos e o usuário desconhece que por trás de tudo que está visualizando um virus acabou de se instalar em sua máquina.

2.4.3 Virus Polimórficos

Ainda não existe uma forma eficaz para se detectar este tipo de virus, eles não tem um padrão a ser identificado. O que se faz é criar um arquivo de vitima e este fica sempre sendo monitorado, mas o bom virus polimorfico 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 auxilio de outra maquina seria inviavel, enquanto o virus se infecta o usuario tentaria localiza-lo e deleta-lo uma guerra perdida.

Capítulo 3

Polimorfismo

3.1 O que é o polimorfismo?

Segundo o dicionário Aurélio[**?**]
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[**? ?**]. Atualmente, todas as empresas de proteção contra cópia usam polimorfismo juntamente com compactação 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[**?**] 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 demostrar o uso 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‘!edqsdrr‘
A vida passa depressa!	23	V7a sv7gvddv7srgerddv6
A vida passa depressa!	24	Y8nq|y8hykky8|}hj}kky9
ABCDEFGHIJKL	13	LONIHKJEDGFA
abcdefghijkl	13	lonihkjedgfa
abc,abc,abc	31	lon!lon!lon
aaaaaaaaaa	54	WWWWWWWWWW

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 significante e um menos significante. 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 significante o A e o menos significante 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[**?**]. 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];
  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
ABCDEFGHIJKL	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 caracter. 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 as parte de uma cadeia mais longa. Entretanto, este padrão pode ser diminuido e talvez até eliminado somando-se ao caracter a posição dele na linha em módulo 255 e fazendo a operação ou exclusivo com o caracter 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 operação ou exclusivo com o caractere anterior pode ser visto abaixo.

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

Note que o único caractere que continua igual é o primeiro pois ele não tem nenhum

anterior para aplicar a operação $\oplus$ e a posição dele é 0. Olhando para a nova cadeia, vamos ver se entendemos o ocorrido analisando os primeiros cinco caracteres codificados pelo novo algorítimo:

P	P % 2	S	C	C_{bin}	C$\oplus$54	V	X	R[P]
0	0	3	a	01100001	01010111	11101010 = 0xEA	11101010	11101010 = 0xEA
1	1	2	a	01100001	01010111	01011101 = 0x5D	01011110	10110110 = 0xB4
2	0	1	a	01100001	01010111	10101011 = 0xAB	10101101	00011001 = 0x19
3	1	3	a	01100001	01010111	10111010 = 0xBA	10111101	10100100 = 0xA4
4	0	2	a	01100001	01010111	11010101 = 0xD5	11011001	01111101 = 0x7D

Para entender a tabela acima:

- P = posição do caracter no vetor de caracteres
- C = carater da posição P
- S = valor usado para shift de bits. É sempre módulo de 3, com valor variando de 3 a 1, de forma decrescente.
- V = P % 2 == 0 ? ROR C, S : ROL C, S. Ou seja, se for posição múltipla de 2 rotaciona à direita, senão rotaciona à esquerda
- X = (V + P) & 255. O caracter codificado na coluna anterior, mais a posição dele na cadeia em módulo 255
- R[P] = P > 0 ? X $\oplus$ R[P-1] : X. Se a posição no for a inicial, aplica ou exclusivo com o caracter codificado no passo anterior.

Note que na coluna **V** temos o resultado da codificação do nosso primeiro algorítimo. À partir da segunda linha a codificação começa a ficar diferente. O segundo caractere gerado na codificação anterior foi **5D**. 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 algorítimo, 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.

Capítulo 4

O vírus polimórfico

Na época do MS-DOS os vírus eram simples e divididos em categorias básicas: infectadores do setor de boot (boot sector[**?**]), infectadores de arquivos .COM e infectadores de arquivos .EXE. Nesta época a vida também era relativamente fácil para os fabricantes de antivírus pois os vírus eram em menor número e a detecção era baseada em assinatura do código malicioso[1]. As atualizações dos antivírus 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[**?**] 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 fosse colocado 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 antivírus não suspeitasse da existência da infecção. Logo, os desenvolvedores de antivírus 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

[1] A assinatura de um vírus é um padrão de bytes que identifica unicamente aquele vírus

somente não estava mais dando certo pois novas variações do mesmo vírus tinham 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 antivírus começaram a desenvolver algorítmos 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, utilizando análise heurística[**? ?**].

Então, os desenvolvedores de vírus perceberam que para evitar a detecção deveriam modificar a aparência do código, surgindo assim os vírus polimórficos. A criação de vírus polimórficos iniciou-se em meados dos anos 1990, com a criação do vírus chamado **1260**[**?**]. Também nesta época, um desenvolvedor de vírus búlgaro, chamado Dark Avenger[**?**] criou um módulo objeto que ele chamou de *Mutation engine*. Este código foi desenhado para ser ligado ao vírus durante a compilação e ser chamado pelo vírus durante o processo de replicação para dar ao vírus a capacidade de mutação a cada nova infecção. Foi uma grande revolução na forma de pensar e construir vírus e um enorme desafio para a indústria de antivírus.

4.1 As partes do vírus polimórfico

Basicamente, um virus polimórfico pode ser dividido em duas partes: o código do vírus propriamente dito e a rotina de descriptografia. O corpo do vírus é criptografado e a cada nova infecção uma nova criptografia é feita, desta forma tornando as variações impossíveis de detectar através de casamento de padrões.

A rotina de descriptografia é responsável por restaurar o código original do vírus e passar o controle de execução a ele. Esta rotina tem que ser gerada pelo gerador do código polimórfico de tal maneira que não seja um código estático pois senão seria facilmente detectável pelos antivírus usando busca por padrões e todo o trabalho da criptografia para esconder o código do vírus seria inútil.

4.2 Protegendo a rotina de descriptografia

Conforme vimos, as técnicas de criptografia são eficientes para proteger o código polimórfico mas possuem um calcanhar de Aquiles: a rotina de descriptografia. Como

o antivírus não poderia detectar o codigo do vírus, uma vez que ele está criptografado, então a única possibilidade de detectar o vírus é através da rotina de descriptografia. Esta é uma grande vantagem, pois o criador do vírus tem certeza de que o código malicioso está protegido pela criptografia e portanto não será detectado, mas também é um grande problema pois o código da descriptografia fica exposto e, em geral, é um código pequeno o que o torna difícil de ser camuflado. Existem várias técnicas usadas para proteger esta rotina contra algoritmos de heurística, casamento de padrões e mesmo engenharia reversa. Vamos ver algumas delas.

4.2.1 Técnicas anti-antivírus

As técnicas anti-antivírus tem a finalidade de impedir a detecção do código malicioso. Algumas destas técnicas, devido ao avanço dos antivírus, não funcionam mais, no entanto foram largamente utilizadas no passado e por isso são descritas nesta seção.

4.2.1.1 Retrovírus

Na natureza, um retrovírus ataca o sistema imunológico. Por este motivo, um vírus que tenta desativar o antivírus que está sendo executado no computador, é chamado de retrovírus. Em geral, a primeira ação do retrovírus, após estar em execução, é listar todos os processos que estão sendo executados no computador, buscando pelos processos que ele reconhece como antivírus e matando-os em seguida. O vírus pode também buscar além de antivírus outros processos de segurança, como firewall e anti-spyware. Algumas vezes o retrovírus pode passar-se por um componente do próprio antivírus e tentar fazer com que o usuário desinstale a proteção[**?**], mostrando uma mensagem, por exemplo, de que o sistema precisa ser atualizado e que o antivírus precisa ser desinstalado e/ou desativado momentaneamente. Algumas vezes ainda, após a desinstalação, ele instala um falso antivírus para evitar que o usuário volte a instalar novamente a sua proteção.

Um retrovírus não precisa necessariamente matar o processo do antivírus ou desinstalá-lo, ele pode simplesmente modificar a prioridade do processo, tornando-a tão baixa que o antivírus poderá nem sequer ser executado.

4.2.1.2 Ocultação do ponto de entrada

Todo executável tem um ponto de entrada. Este ponto de entrada, é onde o código começa a ser executado. O ponto de entrada de um executável é marcado no cabeçalho do arquivo e está descrito no apêndice **??**. Em C, isso corresponderia à função main(). Os primeiros vírus adicionavam o código malicioso ao final do arquivo e mudavam o ponto de entrada no cabeçalho do executável para apontar para o vírus. Outros, adicionavam uma instrução para saltar para o início do código do vírus, sem mudar o cabeçalho do executável. Em qualquer destes casos, o trabalho do antivírus era muito facilitado pois só precisava analisar o início do código do ponto de entrada do executável para determinar se existia ali um vírus conhecido.

Vírus modernos não mudam mais o ponto de entrada no cabeçalho do executável, nem inserem salto no código original. O vírus **Simile**[**?**] é um caso muito interessante. Além de usar um mecanismo polimórfico muito difícil de ser detectado, chamado Tuareg[**?**] ele infecta tanto executáveis do Windows quanto executáveis do Linux. Diferente da abordagem padrão, a execução deste vírus não é durante a inicialização do executável. Ele só será executado quando o programa principal terminar. No windows, ele busca na seção IMPORT do PE a função de API **ExitProcess** ou **_exit**. Se não encontrar, o arquivo não é infectado. Se encontrar, ele muda a chamada à API para chamar diretamente o codigo do vírus. No Linux, ele substitui as chamadas à função **exit**, fazendo com que apontem para o código de inicialização do vírus. Portanto, somente quando o programa terminar é que o vírus será executado.

Técnicas semelhantes podem ser usadas para qualquer API do sistema.

4.2.1.3 Anti-emulação

Antivírus modernos executam o código de programas desconhecidos em uma máquina virtual, ou emulador de execução, chamada SandBox a fim de detectar ações suspeitas de vírus. Em geral, este processo é executado apenas na primeira vez que o usuário for usar o novo programa. Logo, os criadores de vírus criaram técnicas para burlar este processo de emulação.

Uma idéia básica é que o antivírus não pode emular todo o código do executável pois se assim o fizer, irá tomar um tempo inaceitável pelo usuário. Então, o criador do vírus só precisa ter certeza de que terá código suficiente para ser emulado e que pareça legítimo a fim de burlar o emulador.

Outra abordagem seria não executar o vírus todas as vezes, mas sim de forma aleatória. Assim as chances de que o emulador não detecte o vírus são ampliadas pois o código emulado pareceria inofensivo. Digamos, por exemplo, que um vírus executasse somente nas sextas-feiras. Assim, somente seria detectado pelo emulador se a primeira execução do usuário tivesse sido numa sexta-feira. Também poderia usar um contador: a cada 100 execuções, uma dispara o vírus.

Em geral, emulador assume que o código malicioso está próximo do Entry Point, conforme discutimos no item anterior, portanto, técnicas de ocultação do ponto de entrada ainda são válidas com alguns emuladores.

Vírus mais sofisticados conseguem detectar quando estão sendo executados em um emulador e não executam nenhum código anormal neste caso. Simplesmente devolvem a execução para o código legítimo da aplicação.

4.2.2 Técnicas anti-debugging

Quando um anti-virus não consegue reconhecer o vírus, uma amostra do arquivo infectado é enviado para análise por técnicos da empresa de antivírus, que usarão técnicas de engenharia reversa para analisar o funcionamento do código malicioso. Assim que conseguirem entender o funcionamento do vírus, uma nova vacina é criada. Portanto, é de grande importância para o desenvolvedor do código malicioso que este processo seja dificultado ao máximo, prorrogando assim o tempo de vida útil do vírus, worm, malware ou spyware.

A engenharia reversa é largamente utilizada todos os dias, com propósitos nobres e outros não tão nobres assim:

- Entender como funciona um algorítmo que teve o fonte perdido ou cujo fornecedor não existe mais.
- Estudar o código de um driver proprietário que não disponibiliza os fontes e que está com defeito ou que não existe versão para o SO desejado.
- Estudar um código malicioso a fim de criar uma defesa contra o mesmo.
- Estudar um algorítmo para criar uma outra versão e obter lucro vendendo sua própria solução.

Muitas empresas querem proteger seu patrimônio intelectual e empregam, além

de criptografia, técnicas para impedir ou dificultar muito a engenharia reversa em seus produtos. A seguir vamos descrever brevemente algumas das técnicas utilizadas, tendo em mente que várias delas não funcionam nos dias atuais mas vamos escrever sobre elas pois foram muito utilizadas pelos desenvolvedores de vírus.

4.2.2.1 Breakpoint e Single-step

Um software de depuração tem por princípio pode executar o código linha a linha (ou de instrução em instrução assembler) ou passo-a-passo (single step) e ter pontos de parada pré-determinados (breakpoint) onde a execução será pausada para que o programador possa analisar o código/pilha/flags/variáveis.

Nos processadores anteriores ao 80386[**?** , Cap. 12] da intel, o **breakpoint** era feito através da instrução assembler INT 03, cujo código especial era de apenas um byte: 0xCC. Isso facilitava muito pois poderia substituir qualquer instrução do processador. Então o depurador só precisava acrescentar o tratamento apropriado para interceptar a interrupção 03. Este tratamento deveria incluir a restauração do byte original do código sendo depurado. Como um vírus poderia usar este recurso como técnica anti-debugging? Simples: era atrelado à INT 03 uma rotina que descriptografava o código imediatamente seguinte à chamada da INT 03. Exemplo:

```
    .
    .
0200    POP AX
0201    MOV DX, 02
0205    INT 03
0206    XOR AX, BX
0208    ROR CX, 1
020A    IRET
    .
    .
    .
```

Se um depurador estivesse sendo executado, o código pararia em INT 03 e não iria mais para frente pois o depurador entenderia o INT 03 como um breakpoint. Quando rodado fora do depurador, a rotina do vírus seria chamada e descriptografaria o código abaixo de 0206.

Quando é chamada uma interrupção, a rotina recebe na pilha o endereço de retorno da chamada, que aponta diretamente para o próximo código a ser executado.

Note que isso abre um leque enorme de utilização. Em vez de ser código criptografado, os bytes iniciando em 0206 poderiam ser endereços para uma tabela de jumps, que indicaria a próxima posição a ser executada do código.

À partir do 80386, a depuração nativa do processador, incluindo registradores específicos para este propósito, tornou as coisas um pouco mais trabalhosas mas ainda assim o mesmo princípio ainda é válido.

4.3 Polimorfismo em linguagens interpretadas

Vírus de computador podem ser escritos em qualquer linguagem. Alguns vírus são escritos para infectar o próprio código fonte dos programas, antes mesmo deles serem compilados, fazendo assim com que o código malicioso faça parte do código do software legítimo. Este tipo de vírus é bem raro pois é muito difícil analisar o código fonte de um software para saber exatamente onde colocar o código malicioso minimizando as chances de ser facilmente detectado. Um exemplo deste vírus é o **Induc**[**?**]. Surgido em 2009, este vírus tem uma ação bem interessante: caso no computador exista o ambiente de desenvolvimento (IDE) da linguagem Delphi® nas versões de 4 a 7, o vírus modifica a biblioteca básica da linguagem fazendo com que todo e qualquer software que seja produzido neste ambiente tenha uma cópia do vírus. No entanto, o vírus em sí não causa nenhum dano, apenas se propaga para todo o sofware gerado na máquina contendo o Delphi. Apesar do Induc não fazer nenhum mal, a idéia em sí pode ser usada para qualquer fim malicioso. Tanto que em 2011 surgiram duas variações do Induc, chamadas **Induc.B** e **Induc.C**[**?**]. Estas variações são maliciosas e usam técnicas anti-debugging e polimorfismo para ocultar seu código.

No âmbito das linguagens interpretadas, os vírus de macro em Visual Basic for Applications (VBA) que infectavam documentos dos aplicativos do Office da Microsoft, são o exemplo clássico. Talvez um dos vírus mais famosos deste universo tenha sido o Melissa[**?**]. O Melissa infectava arquivos do MS-Word e usava o Outlook para enviar e-mail para todos os contatos da vítima com o fim de propagação. Em 1999 ele chegou a derrubar alguns servidores da internet porque gerou um congestionamento no sistema de e-mail. Existem muito poucos vírus polimórficos em macro. Uma análise mais detalhada deste tipo de vírus pode ser encontrada em [**?** **?**] ou em forma de artigo em [**?**]. Ainda dentro dos códigos interpretados, devemos citar que até mesmo os arquivos .BAT do DOS/Windows podem ser usados como vírus. Um exemplo disso

é o worm Mumu[**?**].

Existem vírus desenvolvidos para linguagens interpretadas, como Python e Ruby e até mesmo JavaScript. No Ruby, por exemplo, o que torna possível a execução do código do polimórfico é a função EVAL(). O polimorfismo em linguagens interpretadas não é necessariamente menos eficiente que dos códigos em assembler mas é diferente uma vez que existe a dependência do interpretador da linguagem. Esta dependência impõe certas limitações, naturalmente, derivadas da estrutura de cada linguagem.

Vamos descrever brevemente algumas das técnicas usadas para fazer o polimorfismo nas linguagens interpretadas. Várias destas técnicas são analisadas em [**?**].

4.3.1 Criptografia

Criptografias simples podem ser feitas no código. Por exemplo, usando a função EVAL do Ruby, é possível fazer uma criptografia super simples, transformando o código fonte do vírus na codificação BASE64. Isso já seria eficiente para esconder o código da maioria dos usuários comuns. Por exemplo, o código abaixo não é nada intuitivo saber o seu conteúdo:

UFJJTlQgIlZPQ8ogRk9JIElORkVDVEFETyEi

Foi codificado em BASE64 usando uma ferramenta online[2]. Se decodificarmos o texto anterior, teremos:

PRINT "VOCÊ FOI INFECTADO!"

Para executar códigos deste tipo basta atribuir o texto codificado à uma variável, decodificar e executar EVAL. Claro, estamos sendo simplistas mas o intuito é demonstrar as possibilidades de forma didática.

4.3.2 Mudança de nome de variáveis

Mudar o nome de variáveis e/ou nome de funções/procedimentos a cada nova geração também é uma técnica que dificulta a identificação do código. As variáveis podem ter nomes com letras randômicas de tamanho fixo ou tamanho variável. O conceito é simples e na prática é bastante utilizada.

[2]Disponível em http://www.motobit.com/util/base64-decoder-encoder.asp

4.3.3 Código inerte

Inserção de código que não faz nada. Por exemplo: **A = A** ou **B = A; A = B**. Ainda pode-se usar loops desnecessários, chamadas a rotinas que não fazem nada. Qual a finalidade deste tipo de código? A de sempre: tornar uma geração diferente da anterior sem modificar o funcionamento do vírus.

4.3.4 Montagem dinâmica de código

Tendo à disposição uma função tipo EVAL(), é possível montar o código que será executado de forma dinâmica. Considere o código abaixo:

Capítulo 5

Conclusão

Referências Bibliográficas

[1] Armadillo Engine. Software para proteção contra cópia e engenharia reversa.. Disponível em http://www.siliconrealms.com/additional-info.php.

[2] EXE Stealth Packer. Software comercial para proteção contra engenharia reversa.. Disponível em http://www.webtoolmaster.com/packer.htm.

[3] IDAStealth plugin. Plugin (freeware e opensource) para proteger o debugger IDA contra várias técnicas anti-debugging, possibilitando desta forma depurar software que usam tais técnicas.. Disponível em http://newgre.net/idastealth.

[4] PELock — software copy protection and license key system. Software para gerar versões demonstrativas de aplicativos comerciais, com proteção contra engenharia reversa.. Disponível em http://www.pelock.com/products/pelock.

[5] Prejuízo causado por cibercrime no Brasil. [Online; acessado em 16-Fev-2013].. Disponível em http://goo.gl/euHpi.

[6] Virus magazines, source code and tutorials. Disponível em http://www.thehackademy.net/madchat/vxdevl/vxmags/.

[7] Como funciona a análise heurística, 2009. [Online; acessado em 16-Fev-2013].. Disponível em http://forums.avg.com/pt-pt/avg-forums?sec=thread&act=show&id=371.

[8] Rogue turning retrovirus, 2010. [Online; acessado em 16-Fev-2013].. Disponível em http://www.symantec.com/connect/blogs/rogue-turning-retrovirus.

[9] 2012 norton study: Consumer cybercrime estimated at $110 billion annually, 2012. Disponível em http://www.symantec.com/about/news/release/article.jsp?prid=20120905_02.

[10] Norton cybercrime report, 2012. Disponível em http://www.norton.com/2012cybercrimereport.

[11] Black Hat USA. Undocumented PECOFF, 2011. [Online; acessado em 16-Fev-2013].. Disponível em http://www.reversinglabs.com/sites/default/files/pictures/PECOFF_BlackHat-USA-11-Whitepaper.pdf.

[12] Xu Chen, J. Andersen, Z.M. Mao, M. Bailey, e J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, páginas 177 –186, junho de 2008.

[13] Frederick Cohen. *Computer viruses*. University of Southern California, 1985.

[14] Frederick Cohen. *A short course on computer viruses*. ASP Press, Pittsburgh, PA, 2 edition, 1990.

[15] Aurélio Buarque de Holanda Ferreira. Dicionário Aurélio online. Disponível em http://www.dicionariodoaurelio.com/.

[16] Mental Driller. Interview with the mental driller/29a, Março de 2002. Disponível em http://www.thehackademy.net/madchat/vxdevl/interviews/mentaldriller%2301.txt.

[17] Nicolas Falliere. Windows anti-debug reference, 2010. [Online; acessado em 16-Fev-2013].. Disponível em http://www.symantec.com/connect/articles/windows-anti-debug-reference.

[18] Peter Ferrie. The ultimate anti-debugging reference, 2011. [Online; acessado em 16-Fev-2013].. Disponível em pferrie.host22.com/papers/antidebug.pdf.

[19] Brent Graveland. The interesting case of the induc virus, 2010. [Online; acessado em 16-Fev-2013].. Disponível em http://www.symantec.com/connect/blogs/interesting-case-induc-virus.

[20] Robert Lipovsky. The induc virus is back, 2011. [Online; acessado em 16-Fev-2013].. Disponível em http://www.welivesecurity.com/2011/09/14/the-induc-virus-is-back/.

[21] Michal Ludvig. Intel 80386 programmer's reference manual, 1986. [Online; acessado em 16-Fev-2013].. Disponível em http://www.logix.cz/michal/doc/i386/.

[22] Mark A. Ludwig. *The Giant Black Book of Computer Viruses*. American Eagle Publications, Inc, Post Office Box 1507. Show Low, Arizona, 1 edition, 1995.

[23] Adrian Marinescu. An analysis of simile, 2003. [Online; acessado em 16-Fev-2013].. Disponível em http://www.symantec.com/connect/articles/analysis-simile.

[24] Gabor Szappanos. Are there any polymorphic macro viruses at all? (...and what to do with them). *Virus Bulletin Conference*, 2002. [Online; acessado em 16-Fev-2013].. Disponível em http://craigchamberlain.com/library/malware/Polymorphic%20Macro%20Viruses.pdf.

[25] Gabor Szappanos. Polymorphic macro viruses, part one, 2010. [Online; acessado em 16-Fev-2013].. Disponível em http://www.symantec.com/connect/articles/polymorphic-macro-viruses-part-one.

[26] Gabor Szappanos. Polymorphic macro viruses, part two, 2010. [Online; acessado em 16-Fev-2013].. Disponível em http://www.symantec.com/connect/articles/polymorphic-macro-viruses-part-two.

[27] Peter Szor. Hunting for metamorphic. Disponível em http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf.

[28] Peter Szor. Virus analysis: The inviršible man. *Virus Bulletin*, Maio de 2000. Disponível em http://www.peterszor.com/invirs.pdf.

[29] Peter Szor. Drill seeker. *Virus Bulletin*, Janeiro de 2001. Disponível em http://www.peterszor.com/drill.pdf.

[30] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, Fevereiro de 2005.

[31] Wikipedia. Mumu (computer worm) — wikipedia, the free encyclopedia, 2009. [Online; acessado em 16-Fev-2013].. Disponível em http://en.wikipedia.org/w/index.php?title=Mumu_(computer_worm)&oldid=314377507.

[32] Wikipedia. 1260 (computer virus) — wikipedia, the free encyclopedia, 2012. [Online; acessado em 16-Fev-2013].. Disponível em http://en.wikipedia.org/w/index.php?title=1260_(computer_virus)&oldid=527495020.

[33] Wikipedia. Dark avenger — wikipedia, the free encyclopedia, 2012. [Online; acessado em 16-Fev-2013].. Disponível em http://en.wikipedia.org/w/index.php?title=Dark_Avenger&oldid=513782286.

[34] Wikipedia. Heuristic analysis — wikipedia, the free encyclopedia, 2012. [Online; acessado em 16-Fev-2013].. Disponível em http://en.wikipedia.org/w/index.php?title=Heuristic_analysis&oldid=529072201.

[35] Wikipedia. Polymorphic code — wikipedia, the free encyclopedia, 2012. [Online; acessado em 16-Abr-2012].. Disponível em http://en.wikipedia.org/w/index.php?title=Polymorphic_code&oldid=487079723.

[36] Wikipedia. Boot sector — wikipedia, the free encyclopedia, 2013. [Online; acessado em 16-Fev-2013].. Disponível em http://en.wikipedia.org/w/index.php?title=Boot_sector&oldid=537371231.

[37] Wikipedia. Computer virus — wikipedia, the free encyclopedia, 2013. [Online; acessado em 16-Fev-2012].. Disponível em http://en.wikipedia.org/w/index.php?title=Computer_virus&oldid=486444876.

[38] Wikipedia. Debugging — wikipedia, the free encyclopedia, 2013. [Online; acessado em 16-Fev-2013].. Disponível em http://en.wikipedia.org/w/index.php?title=Debugging&oldid=533173326.

[39] Wikipedia. Digital watermarking — wikipedia, the free encyclopedia, 2013. [Online; acessado em 16-Fev-2013].. Disponível em http://en.wikipedia.org/w/index.php?title=Digital_watermarking&oldid=537072013.

[40] Wikipedia. Melissa (computer virus) — wikipedia, the free encyclopedia, 2013. [Online; acessado em 16-Fev-2013].. Disponível em http://en.wikipedia.org/w/index.php?title=Melissa_(computer_virus)&oldid=537815781.

[41] Wikipedia. Stoned (computer virus) — wikipedia, the free encyclopedia, 2013. [Online; acessado em 16-Fev-2013].. Disponível em http://en.wikipedia.org/w/index.php?title=Stoned_(computer_virus)&oldid=535508972.

[42] Wikipédia. Sequência pseudoaleatória — wikipédia, a enciclopédia livre, 2012. [Online; acessado em 16-Fev-2013].. Disponível em http://pt.wikipedia.org/w/index.php?title=Sequ%C3%AAncia_pseudoaleat%C3%B3ria&oldid=30480084.

Apêndices

Apêndice A

Estrutura de Arquivos PE

A.1 Arquivo PE

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

A.1.1 Estrutura de arquivo PE.

<table>
<tr><td>DOS 2 - Cabeçalho EXE compatível</td><td rowspan="4">Seção DOS 2.0 (para compatibilidade com DOS somente)</td></tr>
<tr><td>Não utilizado</td></tr>
<tr><td>OEM - Identificador
OEM - Info
Offset para cabeçalho PE</td></tr>
<tr><td>DOS 2.0 Stub Program & Reloc. Table</td></tr>
<tr><td>Não utilizado</td><td></td></tr>
<tr><td>PE - Cabeçalho</td><td>Palavras limitadas a 8 bytes</td></tr>
<tr><td>Tabela de seções</td><td></td></tr>
<tr><td>Image Pages
· Info de Importação
· Info de Exportação
· Info de correção
· Info de recursos
· Info de debug</td><td></td></tr>
</table>

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 executavel 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 versao 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 multiplo 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.

«««< HEAD =======

A.1.3 Cabeçalho

»»»> 156e892e3d131dc73adf19d98edc1e359a8c7b48 Seu cabeçalho descreve partes importantes para o modelo, não é presente ao inicio do arquivo pois está é reservada para compatibilidade com o DOS, que nada mais é que mensagens de aviso que o programa não funciona nesta versão. São alguns campos importantes para este documento:

WORD Machine: Indica a CPU para qual o arquivo foi construido. Diversos virus checam este campo para verificar ase a versão é a mesma para a qual foi projetado. Outros virus não o fazem e infectam qualquer arquivo o que gera erros para a abertura do arquivo quando é executado em uma arquitetura diferente. Este é um risco para virus multiplataforma, poderiam atacar diversas arquiteturas com somente um código.

WORD NumberOfSections: São o número de seções em um executável DLL. Este campo tem diversas funções para um virus. Uma delas é que o virus a incrementa para colocar uma parte de si nesta seção. A tabela de seções é alterada no mesmo instante que ocorre essa mudança. Sistemas baseados em Windows NT aceitam no máximo 96 seções em um arquivo PE, os baseados em Windows 9x não verificam este campo.

WORD Characteristics: Contém as flags de informação do arquivo. A maioria dos virus ««««< HEAD verificam para ter a certeza de que é um executável, outros ,construidos para infectar alguma DLL especifica, para ter certeza que o arquivo é uma DLL. Este campo normalmente ======= verificam para ter a certeza de que é um executável, outros, construidos para infectar alguma DLL específica, para ter certeza que o arquivo é uma DLL. Este campo normalmente »»»> 156e892e3d131dc73adf19d98edc1e359a8c7b4 não é alterado pelos virus.

WORD Magic: O cabeçalho começa inicialmente com um campo "mágico". Os virus o verificam para a certeza de que o infectado é um executável comum e não uma ROM ou outro tipo de arquivo.

DWORD SizeOfCode: Este campo tem o teto do tamanho de todas as seções executáveis. Geralmente os virus não alteram este valor ao adicionar uma nova seção, mas provavelmente isso mude nos virus futuros.

DWORD AddressOfEntryPoint: É o endereço de execução que a imagem inicia. Este valor normalmente aponta para a seção de texto. Campo crucial para maioria dos virus Win32. O campo é alterado pela grande maioria dos tipo de infecções para

o ponto de entrado do código malicioso.

DWORD ImageBase: Quando o ligador cria p executável, assume que a imagem é mapeada em um local espécifico da memória. Este endereço fica armazenado neste campo. Se a imagem pode ser lida de um local especifico, não necessita de realocações pelo carregador. Este campo é utilizado por diversos virus para calcular o endereço de alguns itens, mas não é alterado.

DWORD SectionAlignment: quando o executável é mapeado em memória, cada seção inicia de um endereço virtual que é mútiplo deste valor, de no minimo de 4 kb . Grande parte dos virus o utilizam para o local correto para se colocar, mas não alteram este campo.

DWORD FileAlignment: Nos arquivos PE, os dados em si iniciam de um multiplo deste valor. Os virus não alteram este valor mas usam de forma similar ao comentado acima $SectionAlignment$.

DWORD SizeOfImage: Quando o ligador cria a imagem, calcula o tamanho total das partes da imagem que o carregador deve carregar. Isso inclui o tamanho da região inicial da imagem base através do fim da última seção. No fim da última seção há o teto do multiplicador do alinhamento da seção. Quase todos os arquivos infectados utilizam este valor $SizeOfImage$. Muitos calculam o campo incorretamente, o que causa a impossibilidade de execução sob Windows NT, exatamente pelo motivo de que o Windows 9x não verifica este campo ao executar o arquivo. Para alegria de muitos os criadores de virus não testam muito suas criações. Maior parte dos virus de Windows 9x contém esta falha. Muitos antivirus calculam errado este campo na desinfecção, o que causa um executável que era compátivel com windows NT deixe de ser compátivel quando desinfectado.

DWORD Checksum: Este é o checksum do arquivo. Maior parte dos executáveis registram 0 neste campo. Entretando Dlls e drivers devem tem um checksum diferente de 0. Windows 9x ignora este campo e carrega o arquivo, o que facilita sua infecção no KERNEL32.dll. Este campo é utilizado por muitos virus para representar como marcação, para evitar uma dupla infecção, outros tipos de virus recalcula-o para uma infecção mais efetiva.

A.1.4 Tabela de Seções

O número de entradas da tabela de seções e dado pelo campo de número de seções que está no cabeçalho. A 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 s seções são confirmados pelo ligador de forma crescente e adjacente, e devem sem multiplos 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 multiplo do campo de alinhamento do arquivo do cabeçalho do PE e deve ser menor ou igual ao tamanho virtual.

Offset físico: Offset para apntar a primeira página da seção. É relativo ao inicio do arquivo executavel.

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

A.1.5 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 multiplo da seção de alinhamento.

A.1.6 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 referencias 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 referencia a

DLL. A última entrada é nula o que indica o fim da tabela de diretórios.

A.1.7 Exportação

A informaçãode exportação inicia com a tablela 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 referencias de correção para os pontos de entrada desta imagem.

A.1.8 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.1.9 Recursos

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

A.1.10 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.

Apêndice B

As 10 piores pragas virtuais de todos os tempos

B.0.11 Morris - 1988

Praga do tipo Worm, lançada em 1988 por Robert Morris. Causou um prejuizo entre 10 e 100 milhões de dolares, não era um worm mal intencionado. Foi criado com o intuito de medir o tamanho da internet, mas tinha uma grande falha ele infectava o mesmo computador diversas vezes, causando DOS (Denial of Service).

B.0.12 Melissa - 1999

Virus Criado com o nome de uma dançarina da Flórida, a qual o criador era apaixonado (David L. Smith).Causou um prejuizo de 1 bilhão de dólares. Desligava todos os sistemas de e-mails por onde os e-mails infectados passassem. Foi utilizado inicialmente como arquivos de texto que continham senhas de sites pornográficos.

B.0.13 Code Red

O Code red é um worm que utilizava uma vulnerabilidade de overflow do buffer dos servidores Microsoft IIS e se replicavam para outros servidores de mesmo tipo. Prejuizo de 2 bilhões de dólares. Quando ocorria o estouro de buffer o servidor se desligava o todos os sites armazenados e começavam a exibir a mensagem "Hacked by Chinese!".

B.0.14 CIH - 1998

Também conhecido como Chernobyl e criado pelo Chen Ing Hau, causou um prejuizo entre 20 e 80 milhões de dólares. Foi um dos mais devastadores, pois além de se reproduzir ele destruia todos os dados do computador, algumas vezes até dados da BIOS, transformando os computador em um peso de papel. Seu poder de propagação foi neutralizado por uma atualização da Microsoft, pois atacava somente versões antigas do Windows (9x e Millenium).

B.0.15 Slammer - 2003

Worm que causa a ausência de internet na Coreia do Sul por 12 horas. Assim como o Code Red se aproveita da vulnerabilidade do estouro do buffer, mas agora no Microsoft SQL Server. Após a infecção ele causava Denial of Service fazendo com que os banco de dados não respondessem e causasse uma grande lentidão na internet e se replicava a todos os servidores SQL que tivessem a mesma vulnerabilidade, com efeito cascata os sistemas passavam a não responder.

B.0.16 Nimda - 2001

Não foi determinado qual o prejuizo causado por este worm, havima diversos metodos para espalha-lo, por e-mail, rede, sites e backdoors feitos por outros virus, causou muita lentidão na internet. Por conseguir se espalhar ele foi considerado o worm mais rápido até o momento, em apenas 22 minutos se tornou o vírus mais espalhado do mundo.

B.0.17 Blaster - 2003

Criado pelo grupo hacker Xfocus causou o prejuizo de 2 a 10 bilhões de dólares. Com a intenção de atacar sistemas microsoft Windows, se espalhava com a seguinte mensagem "Billy Gates why do you make this possible? Stop Making money and fix your software!!".

B.0.18 Sasser - 2004

Com prejuizo de 10 milhões de dólares, o worm criado por Sven Jaschan atacou máquinas windows com uma vulnerabildiade de segurança em uma porta de rede que permitia a conexão com outros computadores e se espalhar pela internet, este não foi o motivo de sua fama. Afetou diretamente diversas empresas, como Delta Airlines, quer por se infectar com o virus teve que interromper os vôos, a guarda costeira da Inglaterra teve seus serviçoes de mapas interrompidos e a agência France-Press que também teve comunicações com satélites interrompidas.

B.0.19 Storm - 2007

Este worm usava o modo de propação de e-mails polêmicos ou sensacionalistas como "Genocídio de muçulmanos britânicos"ou "Fidel Castro faleceu". Já possuia recursos melhores que seus antecessores, o Storm contruiu uma verdadeira "botnet", utilizava os computadores infectados para realizar ações programadas pelo worm, como ataques a sites especificos. Havia comunicações entre as máquinas infectadas para melhorar as formas de ataque.

B.0.20 I Love You - 2000

Foi o vírus que trouxe problemas e prejuizos ao redor do mundo, causou um prejuizo entre 5,5 a 8,7 bilhões dólares. Pelo simples motivo do assunto do e-mail ser "eu te amo"foi muito difundido. Em maio de 2000, por volta de 50 milhões de computadores foram infectados, incluidno órgão dos govenos de todo o mundo. Vários deles, como a CIA, tiveram que desligar o sistema de e-mails para diminuir a propagação dele.