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

Agradecimentos **iv**

Resumo **v**

Abstract **vi**

Sumário **vii**

1 Introdução **1**

2 Revisão bibliográfica **2**

2.1 Antivírus 2

2.2 História 2

2.3 Antivirus e SO 4

2.3.1 DOS 4

2.3.2 Windows 5

2.3.3 Linux 5

2.4 Técnicas de detecção 5

2.4.1 Virus de pendrive 6

2.4.2 Virus de macro 6

2.4.3 Virus Polimórficos 6

3 Polimorfismo **8**

3.1 O que é o polimorfismo? 8

3.1.1 Polimorfismo usado de forma legal 8

3.1.2 Como funciona? 9

4 O vírus polimórfico **17**

4.1 As partes do vírus polimórfico 18

4.2 Protegendo a rotina de descriptografia 19

4.2.1 Técnicas anti-debugging 19

4.3 Polimorfismo em linguagens interpretadas 19

5 Conclusão **20**

Referências Bibliográficas **21**

A Estrutura de Arquivos PE e ELF **23**

A.1 Arquivo PE 23

A.1.1 Estrutura de arquivo PE. 23

A.1.2 PE - Cabeçalho 24

A.1.3 Tabela de Seções 24

A.1.4 Páginas de imagem 25

A.1.5 Importação 25

A.1.6 Exportação 26

A.1.7 Correção 26

A.1.8 Recursos 26

A.1.9 Debug 26

A.2 Arquivo ELF 26

A.2.1 A estrutura do arquivo ELF 27

A.2.2 Cabeçalho 27

A.2.3 Identificação 28
A.2.4 Entry Point Address 28
A.2.5 Tabela de Cabeçalhos do Programa (PHT) 28
A.2.6 Tabela de Cabeçalhos de Seção (SHT) 29
A.2.7 Seções Especiais 29
A.2.8 Tabela de Strings 30
A.2.9 Tabela de Símbolos (Symbol table) 31

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 ..., enfim, uma gama enorme de exemplos poderiam ser citados. Dentro deste contexto, os virus de computador (e suas variações) são uma ameaça real à qual todos - direta ou indiretamente - estamos expostos.

Capítulo 2

Revisão bibliográfica

2.1 Antivírus

Os antivirus[6] 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 “Virús 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 virús 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 Thus-Fix 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

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. Critopher 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 facilitar seu crime, 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[1]

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[2]. Atualmente, todas as empresas de proteção contra cópia usam polimorfismo juntamente com compactação

[1] http://www.dicionariodoaurelio.com/

[2] 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[3] 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

[3] 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‘!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[4]. 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];
```

[4]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
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 oper-

ação 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[1]), infectadores de arquivos .COM e infectadores 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[2]. 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[3] 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 anti-vírus não suspeitasse da existência da infecção. Logo, os desenvovedores 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.

[1] http://en.wikipedia.org/w/index.php?title=Boot_sector&oldid=524626394
[2] A assinatura de um vírus é um padrão de bytes que identifica unicamente aquele vírus
[3] http://en.wikipedia.org/w/index.php?title=Stoned_%28computer_virus%29&oldid=532807447

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 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 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[4] [5].

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**[6]. Também nesta época, um desenvolvedor de vírus búlgaro, chamado Dark Avenger[7] 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 anti-vírus.

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 anti-vírus usando busca por padrões e todo o trabalho da criptografia para esconder o código do vírus seria inútil.

[4] http://en.wikipedia.org/w/index.php?title=Heuristic_analysis&oldid=529072201
[5] http://forums.avg.com/pt-pt/avg-forums?sec=thread&act=show&id=371
[6] http://en.wikipedia.org/w/index.php?title=1260_%28computer_virus%29&oldid=527495020
[7] http://en.wikipedia.org/w/index.php?title=Dark_Avenger&oldid=513782286

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 decriptografia. 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-debugging

[8] [9] [10] [11] [12] [13] [14] 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 analisar brevemente algumas das técnicas utilizadas.

4.3 Polimorfismo em linguagens interpretadas

[8] pferrie.host22.com/papers/antidebug.pdf
[9] http://en.wikipedia.org/w/index.php?title=Debugging&oldid=533173326
[10] http://thelegendofrandom.com/blog/archives/2100
[11] http://www.symantec.com/connect/articles/windows-anti-debug-reference
[12] web.eecs.umich.edu/ mibailey/publications/dsn08_final.pdf
[13] research.dissect.pe/docs/blackhat2012-paper.pdf
[14] Software: http://newgre.net/idastealth

Capítulo 5

Conclusão

Referências Bibliográficas

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

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

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

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

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

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

[7] 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`.

[8] 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`.

Apêndices

Apêndice A

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.

<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.

A.1.3 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.1.4 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 multiplo 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 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.1.6 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.1.7 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 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.1.9 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

O formato ELF(Executable and Linkable Format, anteriormente Extensible Linking Format) é um tipo padrão para formato de arquivos executáveis, códigos, bibliotecas compartilhadas e core dumps. Foi facilmente aceito em diversas distribuições de Unix. Em 1999 foi escolhido como o arquivo binário padrão para o sistemas Unix e baseados em Unix. Diferentemente dos executáveis proprietários ele é flexivel e extensivel, e não

está limitado para uma arquitetura especifica. Pode ser adotado por diferentes sistema operacionais e outras plataformas.

A.2.1 A estrutura do arquivo ELF

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

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

A.2.2 Cabeçalho

No cabeçalho de um arquivo ELF, existe uma ordem especifica, já para as seções e segmentos não. Este contém toda a organização do arquivo, a partir dele que podemos ter acesso a outras partes utilizando o offset. Temos as identificações:

e_ident: A identificação do arquivo.

e_type: Tipo de objeto.

e_machine: Arquitetura do arquivo.

e_version: A sua versão.

e_entry: Endereço virtual para ponto de inicio do processo.

e_phoff: O tamanho do cabeçalho em bytes.

e_phentsize: O tamanho de uma entrada no cabeçalho do ELF.

e_phnum: O número de entradas no cabeçalho.

e_flags: São as flags para o processador.

e_ehsize: O tamanho do header ELF em bytes.

e_shoff: O tamanho do cabeçalho da seção.

e_shentsize: O tamanho de uma entrada no cabeçalho da seção.

e_shnum- O entradas no cabeçalho da seção.

e_shstrndx - Indice das seções linkado com a tabela de strings.

A.2.3 Identificação

Nos 4 bytes iniciais do cabeçalho existe a especificação de como interpreta-lo, não considera o sistema que o le e nem o resto do arquivo.

A.2.4 Entry Point Address

No e_entry indica o endereço para onde o sistema irá iniciar a execução dos códigos da seção de texto. Este endereço aponta para o inicio do linkador_$start$ e não para o inicio do sistema que o programador define (main).

A.2.5 Tabela de Cabeçalhos do Programa (PHT)

Descreve a criação do processo para o sistema. É obrigatória para os arquivos executáveis e opcional para os realocáveis. Abaixo segue a descrição de alguns dos identificadores:

p_type: Tipo de segmento e como interpretar a informação.

p_offset: Offset a partir do começo ao primeiro byte do segmento.

p_vaddr: Endereço virtual do primeiro byte do segmento.

p_paddr: Endereço físico, reservado para quando o é utilizado.

p_filesz: Número de bytes do segmento.

p_memsz: Número de bytes do segmento na memória.

p_flags: Flags utilizadas no segmento.

p_align: Valor para alinhamento na memória e no arquivo. Quando 0 ou 1 indica que não é necessário, se o for deve ser positivo e em potência de 2.

A.2.6 Tabela de Cabeçalhos de Seção (SHT)

Descreve as seções. Cada entrada é definida pela seção e possue informações dela. Ela é definida como um vetor, o identificador e_shoffdo cabeçalho que define o offset para localizar o inicio e o e_shentsize do tamanho de cada bloco.

sh_name: Indice para a tabela de string de cabeçalho de seção, descrita acima.

sh_type: Tipo semântico da seção.

sh_flags: Flags da seção (one-bit, 0 ou 1), que descreve seus atributos.

sh_addr: Endereço da seção, caso a seção apareça em memória, senão tem o valor 0.

sh_offset: Indica o inicio da seção no arquivo.

sh_size: Tamanho da seção em bytes.

sh_link: Contém o link para a tabela de cabeçalho da seção.

sh_info: Informação extra, interpretado junto com o tipo de seção.

sh_addralign: Verificação do alinhamento dos blocos definidos por sh_addr, sh_addr deve ser divisivel por sh_addralign.

sh_entsize: Tamanho fixo da tabela de seção, se não for deste tipo terá 0.

A.2.7 Seções Especiais

.bss: Dados não inicializados que são utilizados no programa em memória. Inicia com 0 no inicio do programa, este não ocupa espaço no programa.

.comment: Informações de controle de versão.

.data ou .data1: Dados inicializados utilizados no programa em memória (tem também a .data1)

.debug: Informações para debugar os simbolos.

.dynamic: Informações para linkagem dinâmica.

.dynstr: Strings para a linkagem dinâmica.

.dynsym: Tabela de símbolos da linkagem dinâmica.

.fini: Instruções executáveis para finalização do programa.

.got: Contém a Global Offset Table GOT.

.hash: Hashtable de símbolos.

.init: Instruções para inicialização do programa.

.interp: Caminho para o programa interpretador.

.line: Número da linha $nocdigo$ para debugar.

.note: Informações de formato.

.plt: Contém a Procedure Linkage Tabel PLT.

.relname ou .relaname: Informações para realocação.

.rodata ou .rodata1: Dados para somente de leitura, utilizados em segmento que não permite escrita.

.shstrtab: Nomes das seções.

.strtab: Nomes associados as entradas na tabela de simbolos.

.symtab: A tabela de símbolos.

.text: Instruções executáveis do programa.

A.2.8 Tabela de Strings

Tabela onde armazena strings, terminadas com '
0'. O arquivo objeto utiliza-as para representar os simbolos e nomes das seções. É acessada através de seus indices. O campo sh_name do cabeçalho da seção possue o indice para esta tabela que o é indicado pela e_shstrndx do cabeçalho do programa.

A.2.9 Tabela de Símbolos (Symbol table)

Contém referencias e definições para localizar e realocar no programa.

st_name: Indice para a tabela de strings que contém o nome do símbolo. Se possui 0, o símbolo não tem nome.

st_value: Valor do símbolo.

st_size: Tamanho do símbolo, caso haja algum senão possui o valor 0.

st_info: Tipo do símbolo e atributos.

st_other: Possui o valor 0, sem uma definição.

st_shndx: Indice para o cabeçalho de seção utilizada.