Remoção de Proteções de Acesso a Dados Armazenados em Sistemas Computacionais – Ferramentas e Técnicas

Galileu Batista e Sérgio Xavier

***Abstract*— Media contents analysis is a recurrent aspect in computacional forensics. The most prominent techniques are full indexing and searchs relevant data. However, more and more data are managed by information systems, protected by passwords or other means. In this case, full access to the program offers an integrated view and can be essential for the investigation. This paper discuts several tools and techniques helpful to bypass information system's protection in forensics environment.**

***Index Terms*— Reverse Engineering, Software Tools, Programming.**

I. Introdução

A análise de conteúdo é um problema recorrente nas perícias em mídias computacionais. A indexação e posterior busca por cadeias de texto relevantes são procedimentos típicos nesses casos [1]. Há, porém, cada vez mais situações onde os dados estão organizados e são tratados por sistemas computacionais. Nessas situações é mais eficaz acessar os dados através do próprio sistema, que relaciona os dados e lhes confere significado mais apropriado. Na prática, o acesso a sistemas é, quase sempre, protegido, e, os mecanismos de acesso não disponíveis.

A liberação é um processo de cinco passos: 1) identificação da natureza do código executável do programa, especialmente no tocante ao compilador que o gerou; 2) Análise estática do programa, que permite a compreensão da sua estrutura global – decompiladores podem ser usados nesse passo, tornando o trabalho mais simples; 3) Análise dinâmica que permite envidenciar o fluxo de dados e controle do programa; 4) Localização: reduzir a análise aos pontos de implementação de proteções, e, 5) Modificação do programa para liberar as proteções.

Este artigo formaliza o contexto associado à liberação de proteções de programas binários x86 em ambiente Microsoft Windows®, as ferramentas e técnicas empregadas para esse propósito. A apresentação dessas técnicas na literatura é normalmente dispersa, considerando o potencial uso dessa informação. Trabalhos recentes discutem o tema: em [2] há um uma excelente introdução ao tema, incluindo seus aspectos legais; há também a descrição do uso de algumas técncias em situações hipotéticas, [3] discute conceitualmente o tema, sem apresentar uma metodologia de ação. A principal contribuição desse artigo é apresentar sistematicamente as técncias comumente usadas, mas não largamente documentadas. O texto está organizado como segue: a seção II discute a importância da identificação dos compiladores usados para gerar os programas binários para a seleção das ferramentas de liberação discutidas na seção III. Quando a reversão do código não é possível deve-se liberar as proteções diretamente no binário, empregando técnicas descritas na seção V. Antes da liberação, em si, é necessário identificar onde elas acontecem, o que pode ser conseguido com as técnicas da seção IV. A seção VI apresenta técnicas mais recentes de proteção, cuja liberação pode ser mais complicada. A conclusão está na seção VII.

II. Identificação do Programa Compilado

Arquivos compilados são o foco das liberações de proteções tratadas nesse artigo. Nesse sentido é importante identificar o compilador utilizado para converter o programa fonte em objeto, o que permite o uso de ferramentas específicas para guiar o processo de liberação. Por exemplo, um programa compilado usando *Borland Delphi*® pode ser mais facilmente compreendido se um decompilador *Delphi* for empregado.

A principal ferramenta de identificação de assinaturas de executáveis Windows é o *PEiD*[1]. A prática mostra que, em geral, os programas são resultantes de compiladores para as seguintes linguagens: *C*, *Delphi/C++ Builder*, *Visual Basic* e *Java*®, sendo o *PEiD* a ferramenta de identificação do compilador/ferramenta mais utilizada.

Com o surgimento de sofisticadas técnicas de engenharia reversa de código compilado e visando proteger propriedades intelectuais, desenvolvedores estão, cada vez, mais usando

G. Batista de Sousa é Perito Criminal Federal do Departamento de Polícia Federal, lotado no Setor Técnico Científico da Superintendência Regional em Pernambuco (e-mail: galileu.gbs@ dpf.gov.br). É também Professor do Departamento de Estatística e Informática da Universidade Católica de Pernambuco.

S. A. C. Xavier é Perito Criminal Federal do Departamento de Polícia Federal, lotado no Setor Técnico Científico da Superintendência Regional em Pernambuco (e-mail: sergio.sacx@ dpf.gov.br).

[1] Os autores optaram por não mencionar as URLs relativas a softwares, visto que as mesmas podem ser facilmente localizadas através de ferramentas de busca na Internet.

ferramentas que compactam e/ou cifram o código executável. Como resultado, essas ferramentas constroem um novo executável que contém código para reconstruir, em tempo de execução, o programa original. A técnica dificulta a liberação de proteções, uma vez que não é possível fazer atualizações diretamente no executável original, que se encontra compactado ou cifrado dentro de um novo programa.

Liberar programas ofuscados requer, como primeiro passo, evidenciar trechos de que implementam o processo de reconstrução, separando-os daqueles que formam o executável original em si. Esse procedimento pode ser bastante complexo, especialmente porque o processo de reconstrução pode se dar gradativamente e sob demanda [2].

O *PEiD* reconhece a assinatura da maioria dos ofuscadores e contém plug-ins que reconstroem o original em vários casos. Em outras situações é possível encontrar ou desenvolver ferramentas específicas, fazendo com que a maioria dos programas ofuscada possa ser reconstruída. Os programas encontrados no ambiente de perícias forenses (quando da escrita deste artigo) raramente contêm ofuscação.

III. Ferramentas de Análise do Código Executável

A análise do código executável pode ser feita apenas avaliando Há várias ferramentas para análise e execução do código executável genérico. As mais populares são os *debuggers* e/ou *disassemblers*: SoftIce®, IDA Pro®, WDAsm® e *OllyDbg*. A vantagem do *SoftIce* é a abrangência. O *WDAsm* tem a capacidade de abrir formatos mais antigos. *IDA Pro* e *OllyDbg* se equivalem, porém o segundo é gratuito, tem largo suporte da comunidade e um bom conjunto de *plug-ins*, razão pela qual será discutido nesse texto.

OllyDbg é um *disassembler* com *debugger* integrado que, além das funções típicas, realiza análise do código binário, identifica sub-rotinas e padrões de código gerados por compiladores para comandos estruturados, além de permitir a modificação do código binário durante a execução.

Programas escritos em *Delphi* ou *C++ Builder*® podem ser mais bem analisados usando o *Delphi Decompiler* (*DeDe*). O *DeDe* facilita a compreensão do código e permite a recuperação da aplicação, exceto as rotinas de manipulação de eventos, criadas pelo programador da aplicação, que permanecem em código de máquina. Além disso, o *DeDe* não permite mudanças diretas no código executável. O bom suporte à identificação de cadeias de caracteres e de nomes de funções e métodos da API do *Delphi* permite a localização de pontos de referência das eventuais proteções. Estas referências são fundamentais no processo de liberação, ainda que a liberação em si seja efetuada usando outra ferramenta, como o *OllyDbg*. Saliente-se a existência de um *plug-in* para o *OllyDbg*, denominado GODUP, que realiza funções similares ao *DeDe*.

A análise de programas em Java é simplificada em função do formato do *bytecode* e das características da máquina virtual. A natureza da linguagem, faz com que o *bytecode* possua as referências externas a outras classes expressas como cadeias de texto, permitindo-se identificar rapidamente a cadeia estática de chamada de métodos. Além disso, a JVM, por ser uma máquina de pilha, tem como característica um código de muito fácil leitura por humanos. Essas duas propriedades fazem com que existam decompiladores, por exemplo o *DJ Java Decompiler*, que retornam código fonte de alta legibilidade. Nesse sentido, a análise de proteções de programas escritos em Java pode, normalmente, ser feita no código fonte.

Programas escritos em *Visual Basic* (VB) também podem ser decompilados. Em verdade, o resultado, para as versões mais recentes do *VB*, é similar àquele obtido pelo *DeDe* para programas escritos em *Delphi*. Algumas ferramentas, como o *VBReFormer*, permitem a edição visual das propriedades dos formulários e objetos de gráficos presentes no arquivo executável, permitindo liberar proteções mais óbvias.

A Figura 1 apresenta um diagrama com os passos necessários à identificação do programa e as ferramentas adequadas para a liberação em cada caso. Na Tabela 1 estão sintetizadas características e aplicabilidade das ferramentas apresentadas.

Tabela 1 - Aplicabilidade de ferramentas de liberação.

Ferramenta	Aplicabilidade	Resultado da decompilação	Análise do código Binário
DJ Java Decompiler	Java	Código fonte em Java.	-
Delphi Decompiler (DeDe)	*Delphi* e C++ Builder	*Forms*, *resources* e *callbacks* (em *assembly*).	Reconhece referências a cadeias de *caracteres* e APIs *Delphi*.
VBReFormer	Visual Basic	*Forms*, *resources* e *callbacks* (em *assembly*).	Reconhece referências a cadeias de *caracteres* e APIs do *VB*.
OllyDbg	Qualquer executável (formato PE)	Código *assembly*.	Identifica sub-rotinas, variáveis, construções estruturadas e chamadas à APIs do Windows.

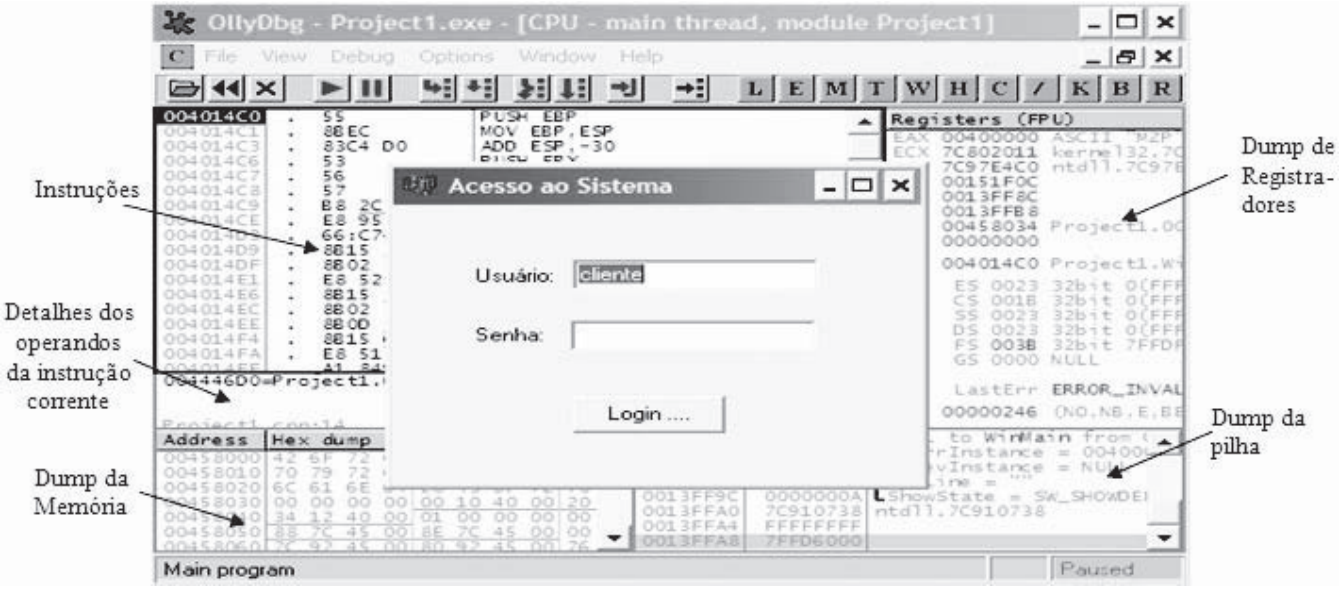

Figura 1 - Visão geral do OllyDbg, com uma aplicação sob *debug*.

IV. Identificação de Pontos de Validação

Considerando a existência de milhares, às vezes milhões, de instruções em um programa e suas bibliotecas associadas, encontrar o(s) ponto(s) que implementam as proteções, ou pontos de validação, é a parte mais complicada do processo de removê-la.

Há duas formas básicas de analisar código, objetivando descobrir pontos de validação: estática ou dinâmica. A análise estática (ou *dead-listing*) é feita seguindo manualmente o código, tentando prever o seu comportamento, sem executá-lo. A análise dinâmica usa um *debugger* para, executando o programa, entender o seu fluxo de dados e controle. Embora forneça uma melhor compreensão do programa, a análise estática é mais demorada e há casos, por exemplo, código ofuscado, onde ela é, praticamente, impossível.

As técnicas de análise dinâmicas variam desde a simples busca por usos de cadeias de caracteres no executável, até a identificação, na pilha de chamadas, de quais rotinas implementam proteções [3].

No contexto em análise, *DeDe* e *VBReFormer* são usados para identificação de *callbacks*. A partir disso e como nos demais casos, o trabalho de liberação é executado utilizando o *OllyDbg*, a única das ferramentas que tem capacidade de *debugging* e modificação do *assembly*, A Figura 1 apresenta a tela do *OllyDbg*, com um programa que lê uma senha e valida contra uma cadeia armazenada internamente, emitindo uma notificação de acerto ou erro. O programa foi escrito e em *C++ Builder* e será usado para as análises subsequentes.

A seguir serão discutidas várias técnicas empregadas para identificação de pontos de validação. Praticamente todos os casos compartilham um princípio: identificar um acontecimento e fazer uma análise da região do código próxima a ele, objetivando compreender o processo de validação. A proximidade diz respeito ao comportamento dinâmico do programa, ou seja, ainda que os trechos de código estejam em endereços muito diferentes, eles devem guardam alguma dependência em tempo de execução.

A. *Acessos a cadeias de caracteres e Chamadas a API*

A forma mais simples de identificar um ponto de validação é através do reconhecimento de instruções onde mensagens de erro ou alerta são emitidas. O *OllyDbg* tem a capacidade de buscar cadeias de caracteres e vincular as instruções que as a referenciam. Forçando paradas (*breakpoints*) em instruções próximas às mensagens pode-se evidenciar as razões que causaram o "erro" e entender a lógica por trás da validação.

Figura 2 - Visualização de instruções que acessam cadeias.

No *OllyDbg*, a visualização das instruções que acessam

cadeias de caracteres é feita através do menu de contexto da janela de instruções, seguido das seleções: (***"search for", "All referenced text strings"***). A Figura 2 mostra o resultado da busca por cadeias. Selecionando a linha **"Senha Incorreta!!!"** encontra o trecho de código mostrado na Figura 4, onde está o ponto de validação da proteção.

Por vezes, as cadeias de caracteres são propositalmente cifradas. Uma alternativa para encontrar pontos de validação é procurar por chamadas à API do Windows que realizam ações de criação de janelas (**CreateWindowEx**) e caixas de mensagens (**MessageBoxA**), exibição de textos (**DrawText**), uso de janelas de diálogo (**CreateDialogEx** e **EndDialog**), entre outras. Essas funções são tipicamente usadas para exibição de mensagens.

Usando a funcionalidade do *OllyDbg* de buscar instruções que referenciam chamadas a bibliotecas (acessível do menu de contexto da janela de instruções, seguindo as seleções: ***"search for", "All intermodular calls"***), é possível aplicar ações semelhantes aos casos onde as cadeias de caracteres não estão cifradas. A Figura 3 mostra a idenficação de **MessageBoxA**.

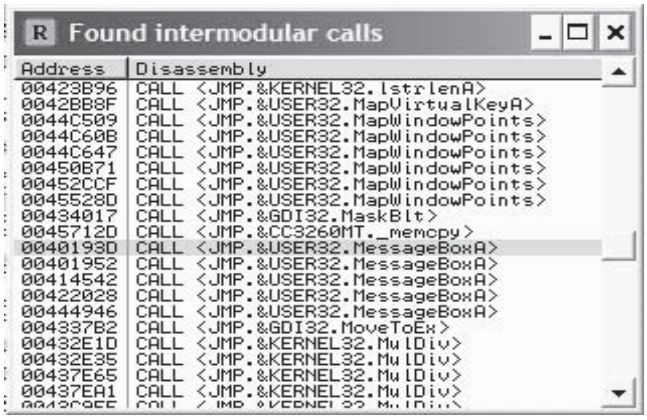

Figura 3 – Visualização de chamadas a bibliotecas.

Ressalte-se que a integração do *debugger* com o *Wingraph32* permite a visualização do fluxograma de uma sub-rotina, bem como dos diagramas de sub-rotinas que a chamam e que são por ela chamadas. Estas funcionalidades simplificam o trabalho de análise da proteção.

É bastante comum que as proteções sejam implementadas utilizando bibliotecas de terceiros. Nesses casos, a procura por cadeias de caracteres ou chamadas a API só deve ser realizada após a carga da biblioteca. Uma estratégia para descobrir o momento exato de realizar a busca é utilizar a funcionalidade ponto de parada após cada carga de biblioteca dinâmica.

B. *Parada em funções de tratamento de eventos*

Quando o acesso a cadeias e APIs é indireto, ou feito por bibliotecas que compõem o ambiente de tempo de execução da linguagem, não é possível usá-las como referências para identificação de pontos de validação. Uma alternativa é forçar paradas no início de todas as sub-rotinas de tratamento de eventos (*callbacks*). Ao executar o programa, pode-se remover as paradas que não se associam com a validação em si. Esse procedimento, típico para programas escritos em *Visual Basic*, permite identificar a rotina que implementa a proteção.

De uma forma geral, a viabilidade dessa técnica reside na capacidade de identificar inícios de sub-rotinas. Como a maioria dos ambientes de tempo de execução segue o que se denomina de "convenções de chamadas de sub-rotinas", o que no ambiente *Windows/x86* significa que as variáveis locais são acessíveis através do registrador "*base pointer*", EBP, é possível supor que uma sub-rotina inicia com **PUSH EPB**. A visualização dessas instruções é feita através do menu de contexto da janela de instruções, seguido das seleções: (***"search for", "All commands"***) e digitando ***PUSH EBP***.

Forçar paradas em todos as *callbacks* pode gerar efeitos ruins num primeiro momento, pois as rotinas de repintura e tratamento de mouse podem ser interceptadas. Contudo, gradativamente pode-se remover as paradas, restringindo-as à *callback* de validação desejada. Dentro da *callback* a análise pode prosseguir para frente, avaliando propriamente a validação.

C. *Memory Breakpoints*

Ocorre freqüentemente que uma notificação de erro foi emitida, tendo sido gerada muitas outras instruções atrás. Nessas situações pode ser difícil encontrar, observando apenas o ambiente próximo à exibição de uma mensagem, o ponto que efetivamente implementa uma validação. Associando paradas em acessos a posições de memória tem-se a oportunidade de identificação o momento de geração da mensagem na memória, tornando a análise mais efetiva e permitindo identificar mais facilmente os pontos de validação, não apenas de exibição de mensagens. No *OllyDbg*, pode-se ativar essa funcionalidade a partir dos menus de contexto das janelas de instrução ou de "*dump* de dados", seguindo: ***"breakpoint", "memory, on access"*** ou ***"memory, on write"***.

Essa técnica é fundamental quando o programa está, de alguma forma, cifrado, e nenhum decifrador está disponível. Com *breakpoints* de memória é possível identificar pontos de escrita no segmento de código, o que revela que o processo de decifragem está em curso. Com critério pode-se selecionar *breakpoints* na instrução que corresponde ao início original do programa (OEP). Descoberto o OEP é um passo importante para recompor o executável original, sem a rotina de decifragem. Pela forma de implementação de *breakpoints* convencionais não é possível usá-los para esse fim.

D. *Saída abrupta*

De alguma forma, as técnicas anteriores baseiam-se em informações oferecidas pelo mecanismo de proteção, como por exemplo: mensagens de erro e telas de registro. Um caso comum ocorre quando, após uma falha na validação da proteção, o programa termina sem quaisquer informações

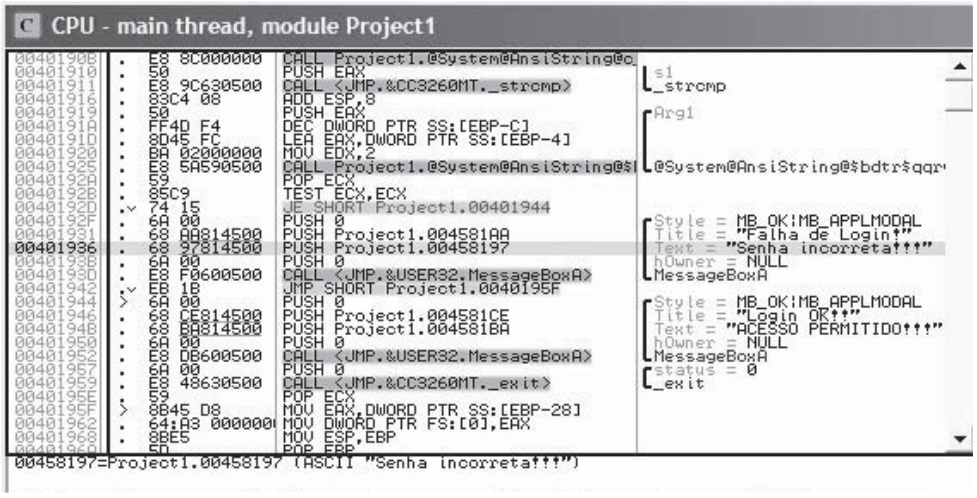

Figura 4 - Ponto de referência da validação da proteção.

adicionais. O tipo de análise adequado à situação é identificar chamada à rotina **`TerminateProcess`** (e suas variações) e acompanhar, via análise da pilha de chamadas, as condições que forçaram a sua execução.

É importante considerar que a chamada que termina o programa não necessariamente ocorre imediatamente depois da falha de validação. Proteções mais sofisticadas introduzem uma aleatoriedade no tempo entre os dois acontecimentos, dificultando a análise. Esse aparente *assincronismo* entre a validação e a notificação ou finalização do programa pode sempre ocorrer. Em geral a solução é identificar a propriedade que ao ser verificada, provoca o final do programa (ou mensagem de notificação), buscando, em seguida, as instruções em que ela é estabelecida. Quando a propriedade é um valor em uma posição de memória, deve-se fazer um monitoramento, usando *memory breakpoints,* dos trechos de código que a atualizam. A análise pode continuar, inclusive com o uso de outras técnicas, até que seja determinado o processo de validação como um todo.

A questão precedente reflete um fato: as técnicas de identificação não são completamente disjuntas, sendo aplicadas concomitantemente em várias situações. Normalmente, pode-se utilizar uma ou mais delas para reduzir o espaço de busca e, em seguida, aplicar outras.

E. Análise da Pilha de Chamadas

Quando as técnicas precedentes não são suficientes, uma técnica importante para reduzir o espaço de busca é a análise da pilha de chamadas. Observando o resultado visível de uma validação (tais como mensagens de erro, construídas dinamicamente) pode-se buscar a seqüência de chamadas de sub-rotinas que culminou com a ação. Avaliando as cadeias de caracteres presentes na pilha, pode-se estimar com precisão, qual das sub-rotinas a escreveu pela primeira vez, o que revela a proximidade do ponto de validação.

Em casos extremos, pode-se marcar pontos de parada em todas as chamadas de sub-rotinas presentes na pilha no momento da exibição da mensagem de proteção. Continuando a execução pode-se concluir que aquelas em que não houve parada não participam da validação. Por outro lado, a última chamada da seqüência em que houve uma parada, está, via de regra, muito próxima do ponto de validação.

O *OllyDbg* identifica pontos de retorno de sub-rotinas, bem como parâmetros de chamadas conhecidas (APIs) e endereços contendo cadeias de caracteres. Esses elementos podem ser visualizados na Figura 1, na janela de *dump* da pilha da Figura 1.

F. Análise para frente

Finalmente, há casos onde nenhuma informação efetiva sobre o ponto de validação pode ser conseguida empregando as técnicas precedentes. Uma análise do fluxo de controle do programa, partindo do seu ponto de entrada rumo às rotinas que implementam as funcionalidades, pode ajudar a compreender a estrutura geral do mesmo, e evidenciar ainda, que sem grande precisão, o ponto de validação.

Em certo sentido a análise para frente é sempre feita. Após identificar a vizinhança da validação, usando outras técnicas, é necessário compreendê-la efetivamente, o que é feito por análise para frente.

V. Remoção de Proteções

As técnicas de remoção da proteção podem ser classificadas em intrusivas e não intrusivas. No primeiro caso o executável é modificado para que a proteção seja removida. No segundo, um algoritmo de geração da senha, ou mesmo a própria senha, é descoberta, podendo ser usada para obter direta ou indiretamente o acesso irrestrito ao sistema.

As técnicas intrusivas mais simples consistem na inversão de uma instrução que verifica uma propriedade, por exemplo, a senha. Na Figura 4 a simples troca do mnemômico ***`JE SHORT Project1.00401944`*** por ***`JNE SHORT Project1.00401944`*** na instrução ***`40192D`*** resulta na liberação da senha de acesso ao programa.

Tabela 2 – Transformações que concretizam a liberação de proteções.

Código Original	Código final	Proteção
`JE END_OK`	`JNE END_OK`	Teste simples de condição que libera a execução completa do programa. Diversas outras condições ocorrem, sendo suficiente invertê-las.
`JNE END_OK`	`JE END_OK`	

`PUSH EBP`
`...`
Código da sub-rotina
`...`
`RETN`
	`MOV EAX, valor`
	`RETN`
		Modificação de sub-rotina de validação para sempre retornar um valor conveniente.

`CMP [MEM],valor` `JZ END_OK`	`MOV [MEM],valor` `JMP END_OK`	Modificação de um valor em memória que é usado em múltiplos pontos de validação. Esta mudança em um ponto, reflete-se em todos os outros.

O *OllyDbg* permite a alteração de instruções em tempo de execução, bastando selecionar (com barra de espaço) uma delas e reescrevê-la. Os devidos ajustes de tamanho também são realizados. Tendo modificação contornado a proteção, é possível construir um novo executável com essa propriedade. É um processo de dois passos:

- No menu de contexto da janela de instruções, deve-se selecionar: ***"Copy to executable", "All modifications"***).
- Como resultado da ação anterior uma nova janela de instruções é criada, na qual deve-se selecionar no menu de contexto a opção ***"Save file"***.

Outra técnica de proteção comum é a validação, em múltiplos pontos, de uma propriedade. Essa proteção é implementada, tipicamente, por testes do valor retorno de uma função de validação. A modificação do valor de retorno libera a proteção nos vários pontos. O mais simples nessa situação é substituir todo o corpo da função de validação por apenas duas instruções: 1***) MOV EAX, valor***; 2) ***RETN***. Aqui se supõe: ***EAX*** deve conter o valor de retorno da função; ***valor*** é um número consistente com a liberação (0 ou 1, normalmente).

Uma situação já discutida trata da validação que consigna seu resultado em uma posição de memória. Posteriormente, e em trechos arbitrários do programa, o valor dessa posição é verificado. A liberação pode ser feita em qualquer dos pontos que ocorrem as verificações. Ou seja, efetuando a substituição de um código da forma ***CMP [MEM],valor; JZ END_OK*** por ***MOV [MEM],valor; JMP END_OK***. Feita a substituição, as outras validações serão todas positivas.

Não são raras as situações onde a validação ocorre em uma *thread*, que é chamada periodicamente para validar ou estabelecer uma propriedade que será validada em outros pontos do programeiro caso é recorrente em programas que usam chaves de *hardware* (*dongles*). Em ambas as situações, uma vez determinados os pontos de validação, a combinação das três técnicas acima é, em geral, suficiente para liberar a proteção.

A Tabela 2 apresenta um resumo das transformações necessárias para liberação do código em cada situação.

VI. TÉCNICAS ANTILIBERAÇÃO

A facilidade de remoção de liberações decorrente do avanço das ferramentas provoca contrapartidas também nas técnicas de proteção. Mesmo quando o código não está ofuscado, há um conjunto de técnicas correntemente em uso para dificultar a liberação das proteções:

- Identificação de presença de *debugger*: a tabela de ambiente de processos (PEB) do Windows armazena informações para caracterizar se um processo está ou não em depuração. A chamada à API **`IsDebuggerPresent`** ou o acesso direto à PEB fornece a informação. Para conveniência, o *OllyDbg* oferece um *plug-in* que "sobrepõe" a API e impede que a condição seja verdadeira.
- Cálculo de verificadores de integridade: é comum fazer verificações se o código do processo foi modificado. Uma função é aplicada sobre valores de posições de memória e testado contra um valor esperado. Em caso de falha o processo é terminado. Quando o resultado da integridade é usado como chave para decifrar um valor, que corresponde a um endereço de um trecho de código a ser executado, pode ser complicado remover a proteção.
- Acesso indireto a bibliotecas do sistema: ocorre quando as chamadas às funções da API estão codificadas por endereço, não por nomes. Essas chamadas implícitas tornam muito mais difícil buscar por pontos com funcionalidades conhecidas.
- Divisão de funções em blocos: é a técnica de subdividir uma função em blocos, colocados em endereços distantes uns dos outros. A execução de cada bloco é ativada por desvios, implementados como chamadas/retornos de função. O código, mais ineficiente, perde em coesão e significado,

ficando pouco aparente a sua real semântica.

- Cifragem e decifragem dinâmicas: quando uma função é crítica em significado, uma proteção eficaz é mantê-la cifrada no programa executável. Para a execução, ela é decifrada, executada e novamente cifrada. Como não permanece na memória, não é possível, estaticamente, definir pontos de parada sobre ela, nem tampouco é eficaz modificá-la. Nesses casos, somente uma análise para frente demorada, com eventual substituição de código pode liberar a proteção.
- Metamorfose de código: a técnica mais sofisticada de antiliberação envolve manter no programa executável um conjunto de rotinas que "reescreve" trechos críticos a cada execução. Entre as mudanças podem estar: renomeamento de registradores, seleção de instruções distintas e divisão de funções em blocos [4, 5, 6].

Algumas das técnicas acima descritas fazem com que o processo de liberação seja difícil o suficiente a ponto de torná-lo inviável. Pode-se dizer, contudo, que no ambiente forense, as técnicas de liberação continuam válidas, pois a maioria dos programas encontrados não tem como foco a proteção em si, mas o seu domínio de aplicação.

VII. Conclusões

Este artigo introduz um conjunto de técnicas e ferramentas usadas para a liberação de proteções. Por se tratar de um tema polêmico, há várias restrições na sua divulgação oficial, tornando difícil a efetiva compreensão do tema. Esse artigo visa contribuir para reduzir essa lacuna e implica, por outro lado, que as técnicas refletem a experiência dos autores na liberação de proteções em vários casos, porém não é possível garantir que sejam as únicas ou mais efetivas.

Finalmente, os autores reconhecem a sensibilidade do assunto analisado e não se responsabilizam, nem apóiam, qualquer uso, implícito ou explícito, das técnicas aqui discutidas, para finalidades ilícitas. Ressaltam também que o texto reflete suas opiniões pessoais, não das instituições a que estão vinculados.

Referências

[1] Galileu Batista, "A Perícia em Mídias de Armazenamento Computacional." Notas de Aula, Academia Nacional de Polícia. 2006.

[2] E. Eilam, "REVERSING: Secrets of Reverse Engineering." Indianapolis: Wiley Pusblishing, Inc. 2005.

[3] G. Hoglund, G. McGraw, "Como quebrar códigos – A arte de explorar (e proteger) software." São Paulo: Pearson Makron Books, 2005.

[4] K. Cooper, L. Torczon, "Engineering a Compiler." New York: Morgan Kauffman, 2003.

[5] M. Jürgen, "Metamorphic Code", disponível online em www.iaik.tugraz.at/teaching/03_advance%20computer%20networks/ss2006/vol12/Metamorphic-code-prenner.pdf

[6] _, "Enhancing software protection with poly-metamorphic code". New South Wales Society for Computers and the Law Journal (6), June 2004.