Иван В. Нечта
Сибирский государственный университет телекоммуникаций и информатики,
ул. Кирова, 86, Новосибирск, 630102, Россия,
e-mail: ivannechta@gmail.com, ORCID iD 0000-0003-0361-2742

АНАЛИЗ УСТОЙЧИВОСТИ ДИНАМИЧЕСКИХ ВОДЯНЫХ ЗНАКОВ

DOI: http://dx.doi.org/10.26583/bit.2017.2.08

Аннотация. В данной работе рассмотрена ранее известная схема внедрения динамических водяных знаков Radix-n, применяемая для борьбы с нелегальным использованием программного обеспечения. Согласно схеме водяной знак представляет собой динамическую структуру данных (граф), которую программа создает в памяти компьютера в процессе выполнения. Топология такого графа определяет скрытую информацию, например, сведения об авторе. Эта информация может быть извлечена для установления авторства в рамках судебных разбирательств. В ходе данного исследования было установлено, что рассматриваемая схема, считавшейся ранее одной из наиболее надежных, обладает рядом особенностей, позволяющих злоумышленнику выявить этап построения водяного знака в программе и, соответственно, его исказить или удалить. Автором данной статьи показан недостаток схемы Radix-n, который заключается в том, что с помощью сведений, полученных от перехватчика Api-функций выделения динамических блоков памяти, можно выявить динамические структуры данных программы, среди которых присутствует водяной знак. Анализ содержимого оперативной памяти позволяет обнаружить указатели на динамически созданные объекты (массивы, переменные, экземпляры классов и т.д.) программы. Различные динамические объекты в памяти, связанные между собой указателями образуют структуры данных программы (списки, стеки, деревья и иные графы), в том числе и водяной знак. В ходе проведенных экспериментов установлено, что в подавляющем числе случаев количество структур данных в программах невелико, что повышает вероятность реализации успешной атаки злоумышленником. В настоящей работе также предложен алгоритм поиска компонент связности графа с линейной трудоемкостью, в случае, когда количество вершин порядка 106. На основе полученных в эксперименте результатов была предложена новая схема внедрения водяных знаков, являющаяся устойчивой к разработанной атаке. В предлагаемой схеме используется граф для хранения водяного знака, где связи между узлами реализованы с помощью уникальных сигнатур. Предложено использовать шифрование содержимого узлов графа (кроме сигнатур) для того, чтобы злоумышленник не смог выявить наличие связи между узлами.

Ключевые слова: динамические водяные знаки, стеганография исполняемых файлов.

Для цитирования. НЕЧТА, Иван В. Анализ устойчивости динамических водяных знаков. Безопасность информационных технологий, [S.l.], v. 24, n. 2, p. 72-81, june 2017. ISSN 2074-7136. Доступно на: <https://bit.mephi.ru/index.php/bit/article/view/107>. Дата доступа: 23 june 2017. doi:http://dx.doi.org/10.26583/bit.2017.2.08.

Ivan V. Nechta
Sibirian state university of telecommunications and informatic sciences,
Kirova st., 86, Novosibirsk, 630102, Russia,
e-mail: ivannechta@gmail.com, ORCID iD 0000-0003-0361-*2742*

Robustness analysis of dynamic watermarks

DOI: http://dx.doi.org/10.26583/bit.2017.2.08

Abstract. In this paper we consider previously known scheme of dynamic watermarks embedding (Radix-n) that is used for preventing illegal use of software. According to the scheme a watermark is dynamic linked data structure (graph), which is created in memory during

program execution. Hidden data, such as information about author, can be represented in a different type of graph structure. This data can be extracted and demonstrated in judicial proceedings. This paper declared that the above mentioned scheme was previously one of the most reliable, has a number of features that allows an attacker to detect a stage of watermark construction in the program, and therefore it can be corrupted or deleted. The author of this article shows the weakness of Radix-N scheme, which consists in the fact that we can reveal dynamic data structures of a program by using information received from some API-functions hooker which catches function calls of dynamic memory allocation. One of these data structures is the watermark. Pointers on dynamically created objects (arrays, variables, class items, etc.) of a program can be detected by content analysis of computer`s RAM. Different dynamic objects in memory interconnected by pointers form dynamic data structures of a program such as lists, stacks, trees and other graphs (including the watermark). Our experiment shows that in the vast majority of cases the amount of data structure in programs is small, which increases probability of a successful attack. Also we present an algorithm for finding connected components of a graph with linear time-consuming in cases where the number of nodes is about 106. On the basis of the experimental findings the new watermarking scheme has been presented, which is resistant to the proposed attack. It is offered to use different graph structure representation of a watermark, where edges are implemented using unique signatures. Our scheme uses content encrypting of graph nodes (except signature), so an attacker is not able to reveal links between nodes and find the watermark.

Keywords: dynamic watermarking, steganography of executables.

For citation. NECHTA, Ivan V. Robustness Analysis of Dynamic Watermarks. IT Security (Russia), [S.l.], v. 24, n. 2, p. 72-81, june 2017. ISSN 2074-7136. Available at: <https://bit.mephi.ru/index.php/bit/article/view/107>. Date accessed: 23 june. 2017. doi:http://dx.doi.org/10.26583/bit.2017.2.08.

Введение

Современные производители программного обеспечения большое внимание уделяют методам защиты программ от нелегального использования. Под «нелегальным» понимается использование программы без покупки лицензионного ключа. Существуют множество методов защиты программы от дизассемблирования или редактирования, например [1-3], однако злоумышленнику зачастую удается найти место в программе, где производится проверка правильности лицензионного ключа и либо исказить алгоритм проверки, либо создать простейший генератор ключей (т.н. keygen). В результате успешной атаки злоумышленник выкладывает в общий доступ программу, лишенную всякой защиты от нелегального использования.

Еще одной угрозой, с которой сталкивается производитель, является незаконное использование программы или ее компонентов недобросовестными конкурентами в составе чужих программ (аналог плагиата). Зачастую автору бывает трудно доказать некорректное заимствование компонентов своих программ в чужом программном продукте.

Авторы статей [4-5] предлагают, чтобы для защиты от подобных проблем производитель встраивал некую секретную информацию (цифровой водяной знак, далее ЦВЗ) в свой продукт, которая содержит сведения об «истинном» авторе. В случае обнаружения некорректного заимствования эта информация может быть извлечена и использована в ходе судебных разбирательств (см. работу [6]).

Очевидно, что водяной знак должен обладать некоторой устойчивостью[1], чтобы злоумышленник не смог его исказить или удалить из программы. В ряде случаев производители явно показывают свое авторство, например, в виде строк об авторстве в меню «О программе», в таком случае злоумышленнику легко найти и исказить ЦВЗ. В

[1] Здесь и далее под «устойчивостью» понимается свойство, обеспечивающее невозможность обнаружения или искажения ЦВЗ в программе.

других случаях информация может быть встроена в особом виде, скрытом от стороннего наблюдателя, когда злоумышленник не может гарантировать ни наличия, ни отсутствия ЦВЗ. Рассмотрим более подробно методы внедрения скрытых ЦВЗ.

Существующие ЦВЗ можно условно разделить на две основные категории:

- статические ЦВЗ, описанные в работах [7-9], представляют данные в виде констант или инструкций программы, фрагментов неиспользуемого кода (т.н. «мертвый код»);
- динамические ЦВЗ, предложенные в работе [10], представляют данные в виде динамической структуры данных, создаваемой в оперативной памяти компьютера на этапе выполнения программы.

Анализ статей [11-13] показал, что наиболее устойчивыми к обнаружению являются динамические ЦВЗ, т.к. анализ работы программы производится пошаговым выполнением команд (т.н. трассировкой), что занимает достаточно большое время. В таких условиях считается затруднительным выявление динамической структуры данных в памяти, отвечающей за водяной знак.

В работе [10] предлагается схема Radix-n динамического ЦВЗ, представленная на рис. 1. Структура водяного знака представляет собой граф. Имеется ключевая вершина, по которой этот граф может быть найден в памяти (например, содержит некоторую сигнатуру, известную только автору). Топология графа определяет содержимое ЦВЗ.

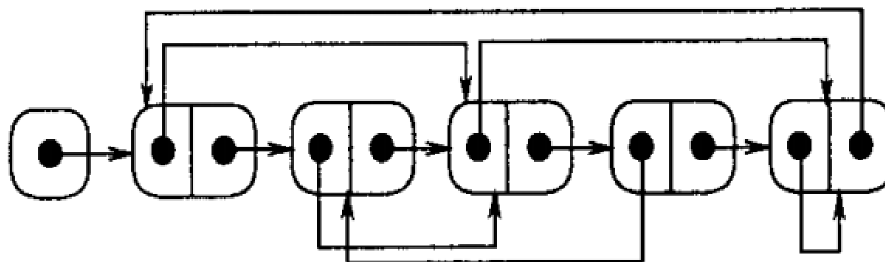


Рис. 1. Цифровой водяной знак по схеме Radix-n
Fig. 1. A digital watermark according to the scheme Radix-n

В той же работе рассматривались возможные атаки, но они требовали наличие исходных кодов программ.

Строго говоря, существует два типа атак:

- статические, которые имеют дело с анализом исходных кодов программ или, что чаще встречается, с анализом через дизассемблирование программы;
- динамические, которые предполагают анализ работы программы в ходе ее выполнения.

Считается, что статические атаки являются менее эффективными, чем динамические. Чаще всего злоумышленник имеет дело с готовым исполняемым файлом (исходные коды ему не доступны[2]) и статический анализ через дизассемблирование не дает полной картины происходящих процессов в программе. Динамический анализ, с одной стороны, дает качественную картину действий программы (Api вызовы, доступ к жесткому диску или сети), и с другой стороны, позволяет отсеять большой объем неиспользуемого кода в самом начале анализа.

По утверждениям авторов статьи [10], в процессе работы программы найти место построения ЦВЗ не представляется возможным, т.к. для этого требуется проанализировать содержимое всей динамической памяти. Во-первых, не ясно как отличить указатели от

[2] Речь идет только о файлах формата portable executable (exe, dll, …). Программы, выполняемые на Java Virtual Machine, с легкостью дизассемблируются и восстанавливаются до высокоуровневого исходного кода.

иных данных, а во-вторых, программа может создавать множество аналогичных структур данных, неотличимых от водяного знака, что усложняет задачу анализа.

В настоящей статье предлагается эффективный метод поиска ЦВЗ в исполняемых файлах. Показано, что схема Radix-n имеет ряд недостатков, которые способствуют в подавляющем большинстве случаев обнаружению времени и места построения ЦВЗ в программе.

Описание предлагаемого анализа.

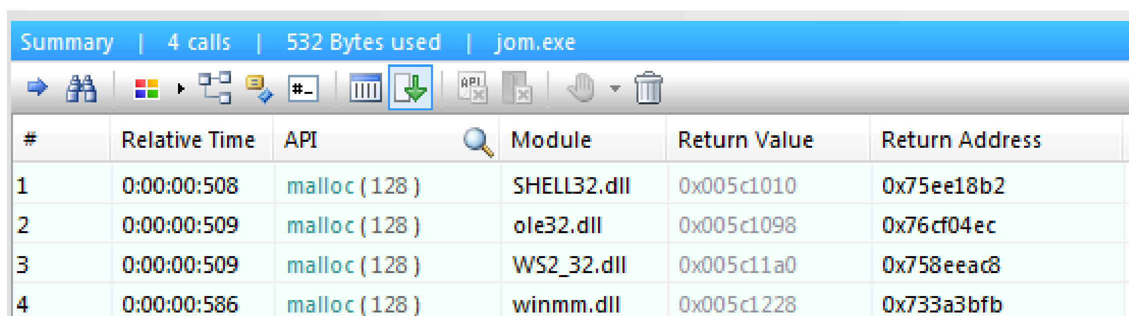
Основная задача, которая ставится в рамках реализации атаки на ЦВЗ, состоит в нахождении этапа в работе программы, во время которого производится построение водяного знака. Выявив указанный этап, злоумышленник с легкостью определит участок кода, ответственный за построение ЦВЗ и произведет его искажение. Наличие дополнительных механизмов контроля целостности (контроль отсутствия изменений) программы, так и противодействие им, не является предметом данного исследования и должно рассматриваться отдельно.

Рассмотрим следующий пример. Допустим, имеется программа, которая при запуске создает ЦВЗ указанным в работе [10] способом (Radix-n), и затем, продолжает работу. Для обнаружения указанного этапа работы программы злоумышленнику потребуется найти в памяти все графоподобные структуры данных, причем узлы графа представляют собой динамически выделенные области памяти, а дуги представлены в виде указателей.

Будем утверждать, что если количество таких структур, создаваемых программой, менее определенного числа $Count_{gr}$, то злоумышленник может без особых усилий выявить среди них ЦВЗ, при наличии полных сведений о структурах. Предлагаемый метод анализа эффективно может работать при $Count_{gr} = 20$, что связано со сложностью редактирования и анализа программ, позволяющее выполнить искажение ЦВЗ. В ситуации, количество указанных структур более 20, предлагаемая в данной работе атака не реализуема. Также будем называть ЦВЗ программы *графом*, т.к. он представлен в виде ориентированного графа и расположен в динамической памяти процесса.

Теперь определим способы получения сведений о создаваемых программой структур данных в памяти. Известно, что для выделения динамической памяти программисты используют специальные Api-функции. В данной работе рассматривается функция *malloc* библиотеки языка C/C++. Стоит отметить, что иные функции (*calloc*, *new* и т.д.) в свою очередь вызывают *malloc*. Будем использовать перехватчик (см. рис. 2) вызовов Api-функций для получения следующей информации:

- время вызова, определяющее этап работы программы;
- размер выделенной памяти;
- адрес выделенной памяти (необходим для последующего анализа содержимого);
- имя модуля запросившего память (это позволит отсеивать ненужные вызовы системных библиотек Winodws, не относящихся к ЦВЗ).



Summary | 4 calls | 532 Bytes used | jom.exe

#	Relative Time	API	Module	Return Value	Return Address
1	0:00:00:508	malloc (128)	SHELL32.dll	0x005c1010	0x75ee18b2
2	0:00:00:509	malloc (128)	ole32.dll	0x005c1098	0x76cf04ec
3	0:00:00:509	malloc (128)	WS2_32.dll	0x005c11a0	0x758eeac8
4	0:00:00:586	malloc (128)	winmm.dll	0x005c1228	0x733a3bfb

Рис. 2. Перехват вызовов Api-функций программы
Fig. 2. Intercepts calls to the Api functions of the program

Получив все необходимые сведения об узлах графа, рассмотрим способ поиска дуг (указателей). Согласно схеме (рис. 1.) ЦВЗ представляет собой отдельный граф, имеющий одну компоненту связности. Указатели в памяти могут быть найдены следующим образом. Пусть $M = \{m_1, m_2, \dots, m_N\}$ последовательность байт участка памяти, N – размер этого участка в байтах. Указатель представляет собой четверку последовательных байт $\{m_i, m_{i+1}, m_{i+2}, m_{i+3}\}$, где $i \in [1; N-3]$, причем указывать он должен на одну из выделенных ранее областей памяти. В текущей реализации в качестве указателей брались такие, которые указывали не только на начало динамически выделенных блоков памяти, но и в произвольное место (внутри) данных блоков.

Таким образом, во всех блоках памяти, выделенных динамически, производится поиск указателей и выявляются связанные структуры данных.

Постановка эксперимента.

В рамках эксперимента планируется показать, что в подавляющем большинстве программ количество связанных структур данных в памяти процесса не превосходит указанной границы $Count_{gr}$. Для анализа содержимого памяти процесса использовался Дамп[3] памяти (подробнее см. работу [14]), который в дальнейшем анализировался. Программа запускалась и завершалась через 10 секунд после полной загрузки.

Было установлено, что процесс выделения памяти (по количеству и объему) идет наиболее интенсивно именно в указанном периоде. Соответственно, построение ЦВЗ разумнее выполнять на данном этапе работы программы. В действительности построение может происходить на любом этапе, но как было установлено в экспериментах, количество создаваемых структур в этот период будет много меньше, чем при запуске и это будет значительно облегчать задачу злоумышленнику.

Будем рассматривать выделяемые программой в памяти блоки данных как некий глобальный граф. Одной из задач, решаемых в рамках данного исследования, является поиск компонент связности глобального графа. При хранении глобального графа в виде матрицы поиск компонент связности может осуществляться с помощью ранее известных алгоритмов [15]. В нашем случае, когда количество узлов глобального графа может достигать порядка 10^6, хранение и обработка графа в виде двумерного массива невозможна.

Рассмотрим схему хранения и поиска компонент связности графа, разработанную автором настоящей статьи и используемую в данном эксперименте. На рис. 3 показан ориентированный граф из 6 узлов.

Каждой вершине соответствует свой элемент (показан прямоугольником), содержащий номер группы. Изначально число групп равно числу узлов графа. Просматривая каждый узел слева направо объединим две соответствующих группы в одну, если текущий узел А имеет связь с узлом Б и дуга идет в направлении от А к Б.

Объединенная группа получает номер соответствующий минимальному из номеров объединенных групп. После однократного прохода по каждой вершине, мы получаем узлы, объединенные в группы. Количество объединяющих групп равно числу компонент связности глобального графа. В итоге мы получаем алгоритм поиска компонент связности, имеющий линейное время работы и не требующего большого объема памяти.

Ниже идет таблица полученных результатов. Здесь «искомым графом» называется компонента связности, состоящая из двух или более узлов (среди них может быть ЦВЗ). Ранее мы уже говорили о том, что если искомых графов будет менее чем $Count_{gr} = 20$, то злоумышленник без труда сможет найти и исказить водяной знак.

[3] Дамп – сохраненное в виде файла содержимое оперативной памяти процесса в некоторый момент времени выполнения (в нашем случае на 10 сек. после загрузки).

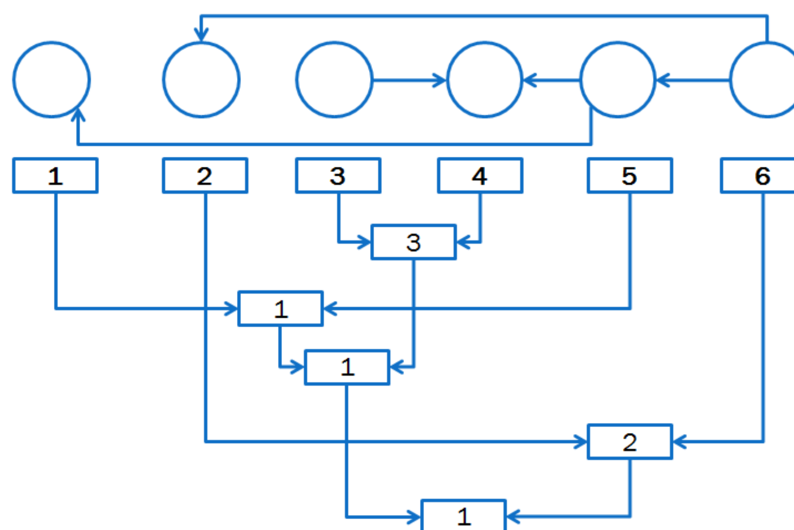


Рис. 3. Схема хранения и поиска компонент связности графа
Fig. 3. The scheme of storage and retrieval of connected components of the graph

Таблица 1. Результаты экспериментальных исследований

Название программы	Количество		
	узлов в глобальном графе	не изолированных узлов глобального графа[4]	искомых графов
WinRar (v.5.00)	159	109	4
Far (v. 3.0.4438)	26	0	0
Gimp (v. 2.8.18)	768667	55412	5030
Chrome (v. 55.0.2883.87)	5222	4116	4
TeXworks (v. 0.4.4)	42737	23523	1655
Word 2007	86	40	2
Excel 2007	52	1	1
Power Point 2007	45	1	1
QtCreator (v. 4.1.0)	466	170	5
Notepad ++ (v. 7.2.2)	22	1	1

Из таблицы видно, что в подавляющем большинстве программ искомых графов менее $Count_{gr}$, это означает, что ЦВЗ может быть обнаружен при помощи предлагаемой атаки. В ходе анализа представленного ряда программ было установлено следующее:

1. При многократных запусках одной и той же программы последовательность вызовов Api-функций оставалась неизменной. Эта особенность дает злоумышленнику возможность отслеживать этап построения ЦВЗ с последующим его искажением.
2. Более 90% блоков памяти (по количеству) выделяется на этапе запуска программы, что делает этот этап работы приложения перспективным для построения ЦВЗ.
3. Содержимое Дампа, за исключением адресов (указателей), практически не меняется от запуска к запуску.
4. Адреса указателей выровнены на границу в 4 байта. Как правило, это связано с тем, что компиляторы по умолчанию осуществляют такое выравнивание (хотя существует способы это выравнивание отключить для оптимизации расходов памяти).

[4] Здесь посчитаны узлы, имеющие хотя бы одну исходящую дугу.

5. При выходе из функции программы, в которой создавались локальные динамические объекты, будет производиться их удаление (для предотвращения так называемой «утечки памяти»). Соответственно, в нашем анализе мы обнаружим, что после запроса на выделение динамической памяти сразу же следует запрос на ее освобождение. В этой связи в Таблице 1 появляется множество дополнительных узлов, которые могут иметь связи. (Реальное количество существующих узлов значительно меньше, например, для Gimp меньше в 7 раз).

Рассмотрим более подробно работу программы Gimp, имеющей наибольшее количество узлов из анализируемого ряда программ. Проведем дополнительное отсеивание узлов, которые, во-первых, уже были освобождены вызовом Api-функции *free*, а во-вторых, имеют размер узла соответствующий атакуемой схеме ЦВЗ (см. рис. 1., размер узла равен 8 байт, т.к. всего два указателя размером по 4 байта).

Таблица 2. Результаты дополнительного отсева узлов

Название программы	Количество		
	узлов в глобальном графе	не изолированных узлов глобального графа	искомых графов
Gimp	2632	16	8

После дополнительного отсеивания мы также видим, что количество искомых графов менее $Count_{gr}$.Таким образом, можно утверждать, что предложенная в работе [10] схема Radix-n для внедрения динамического ЦВЗ не является устойчивой.
Стоит отметить, что в общем случае мы можем продолжать производить отсев узлов, например, по имени модуля (отсекая вызовы от известных системных библиотек от анализа) или иным способом.

Обсуждение полученных результатов.

В предыдущей главе было показано, что применение схемы Radix-n для внедрения ЦВЗ не гарантирует его устойчивости. В нашем эксперименте мы выявляли графы похожие на динамический ЦВЗ. В ходе эксперимента было показано, что при помощи перехвата Api-функции выделения памяти и поиска указателей злоумышленник может существенно сократить время поиска ЦВЗ. Более того, здесь может быть дополнительно произведен отсев найденных графов по топологии графа. Так, часто используемые в программировании графы типа список, стек, дерево и т. д. могут быть исключены из рассмотрения, как не соответствующие схеме Radix-n, что существенно повышает вероятность успешной атаки на водяной знак.

Очевидно, что рассмотренная схема построения ЦВЗ может быть модифицирована таким образом, чтобы в памяти вместо указателя хранилось некоторое смещение. То есть указатель представлен в виде: Указатель = Константа + Смещение. Смещение хранится в памяти, а Константа может рассчитываться программой. Такая схема, на первый взгляд, не позволяет искать связанные структуры, относящиеся к ЦВЗ. Однако при детальном рассмотрении выясняется, что злоумышленник может успешно определить место хранения указателей (ребер графа), относящихся к ЦВЗ следующим образом.

Шаг 1. Запустить программу и получить дамп памяти D_1;

Шаг 2. Повторно запустить программу и получить дамп памяти D_2;

Шаг 3. Получить Дамп разницы D_3 путем побайтового вычитания дампов D_1 и D_2;

Шаг 4. Отсеять указатели на существующие объекты в памяти. Оставшиеся указатели будут относиться к ЦВЗ.

Теперь рассмотрим действия злоумышленника более подробно. В предыдущей главе мы упоминали про то, что содержимое Дампа, за исключением адресов (указателей),

практически не меняется от запуска к запуску. Известно, что все динамические объекты хранятся в специальной области памяти. Адрес указанной области определяется ОС Windows и при каждом новом запуске программы может меняться. Адреса указателей, являющихся суммой адреса начала динамической области памяти и смещения внутри нее меняются от запуска к запуску программы. Соответственно на шаге 3 мы получим дамп D_3, содержащий только указатели, все остальные элементы динамической памяти (т.к. они одинаковые в D_1 и D_2) обнулятся.

Большая часть указателей не относится к ЦВЗ и указывают на реально существующие объекты в динамической памяти. Такие указатели можно успешно выявить и отсеять. Оставшиеся указатели полностью будут относиться к ЦВЗ, что даст злоумышленнику возможность успешно реализовать атаку.

Таким образом, в данной работе было показано, что внедрение динамического ЦЗВ по схеме Radix-n не является устойчивой. Более того модификация связанная с хранением смещений вместо указателей также не повышает устойчивости.

Предлагаемая схема внедрения ЦВЗ.

В предыдущих главах было показано, что исходная схема внедрения ЦВЗ Radix-n позволяет злоумышленнику отслеживать водяной знак в программе. В данной главе будет предложена новая схема внедрения ЦВЗ, которая будет устойчивой к атаке, предложенной в настоящей работе.

Новая схема внедрения предполагает хранение ЦВЗ в виде графа. В отличие от Radix-n узлы графа имеют такой размер, который наиболее часто используется в программе. Рассмотрим рис. 4, на котором показан график зависимости количества выделяемых блоков памяти от его размера. Здесь приводятся данные для программы Gimp.

Из графика мы видим, что в программе чаще всего (10725 раз) выделяются блоки памяти размером 18 байт. Следовательно, размер узлов графа, хранящего ЦВЗ должен быть таким же. Это позволит избежать поиска узлов графа, хранящего ЦВЗ, по заранее известному размеру (в схеме Radix-n размер узла составляет 8 байт).

Вторым отличием предлагаемой схемы внедрения является способ представления данных в ЦВЗ. Radix-n предполагает хранение данных в виде топологии графа, однако в нашем случае ЦВЗ может храниться в виде зашифрованной последовательности байт в каждом узле графа.

Рис. 4. График зависимости количества блоков памяти от его размера
Fig. 4. The graph of the number of the blocks memory of its size

Третьим отличием нашей схемы является способ связи узлов графа. В связи с тем, что описанная в данной работе атака базируется на поиске связанных указателями узлов графа, предлагается связь реализовывать через специальные сигнатуры. Сигнатурой называется уникальная последовательность байт (фиксированного размера), располагающаяся в начале выделенного блока памяти. Уникальность сигнатуры проверяется на этапе внедрения ЦВЗ. Таким образом, в узле графа, хранящего ЦВЗ, содержится следующая информация:

- сигнатура текущего узла;
- полезные данные ЦВЗ (например, сведения об авторе);
- сигнатура следующего узла (заменяющая собой указатель).

Шифрованию подвергаются полезные данные ЦВЗ и сигнатура следующего узла. Наличие шифрования гарантирует, что злоумышленник не сможет отследить связанность узлов графа. Отказ от использования указателей объясняется тем, что при каждом запуске программы указатели могут менять свое значение, соответственно шифрование должно производиться при каждом запуске программы. Считается недопустимым хранение пароля для шифрования в самой программе. В случае с сигнатурами, шифрование производится единственный раз на этапе внедрения ЦВЗ. Таким образом, пароль в программе хранить не требуется.

В данной работе мы рассмотрели известную схему внедрения Radix-n. Показано, что с помощью метода, предложенного в данной работе, злоумышленник может выявлять ЦВЗ, хранящийся в программе. Также была предложена новая схема внедрения ЦВЗ, которая является устойчивой к данной атаке.

СПИСОК ЛИТЕРАТУРЫ:

1. Алейников С. И., Богатов А. О. Защита программ от дизассемблирования. Труды Института системного программирования РАН. 2006. Т. 11. С. 9-16.
2. Collberg C., Thomborson C. and Low D. A Taxonomy of Obfucating Transformations,Technical Report N148, Department of Computer Science. The University of Auckland, July, 1997.
3. Никольская К. Ю., Хлестов А. Д. Обфускация и методы защиты программных продуктов. Вестник УрФО. Безопасность в информационной сфере. 2015. № 2. С. 7-10.
4. Yarmolik V.N., Portyanko S.S. State of the ART in Software Ownership Protection. Computer Information Systems and Industrial Management Application Editors: Khalid Saeed, Romuald Mosdorf, Olli-Pekka Hilmola. Bialystok, Poland. 2003. pp. 188-195.
5. Collberg C.S., Thomborson C. Watermarking, Tamper-proofing and Obfuscation – Tools for Software protection. Computer Science Technical Report #170. University of Auckland, Aucland, New Zealand. February 10, 2000.
6. Козубенко Ю. В. Защита авторских прав на программы для ЭВМ в уголовном, административном и гражданском судопроизводстве. Wolters Kluwer. Russia, 2009, 344 с.
7. Alexander G. Gounares Fingerprinting Executable Code. Patent US20120317421 A1, date 19.07.2012.
8. Anckaert B. et al. Steganography for executables and code transformation signatures. International Conference on Information Security and Cryptology. Springer Berlin Heidelberg. 2004. pp. 425-439.
9. Zaidan A. A. et al. Novel approach for high (secure and rate) data hidden within triplex space for executable file. Scientific Research and Essays. 2010. vol. 5. no. 15. pp. 1965-1977.
10. Collberg C., Thomborson C. Software watermarking: Models and dynamic embeddings. Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM. 1999. pp. 311-324.
11. Moser A., Kruegel C., Kirda E. Limits of static analysis for malware detection. Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual. IEEE. 2007. pp. 421-430.
12. Ernst M. D. Static and dynamic analysis: Synergy and duality. WODA 2003: ICSE Workshop on Dynamic Analysis. 2003. pp. 24-27.
13. Linn C., Debray S. Obfuscation of executable code to improve resistance to static disassembly. Proceedings of the 10th ACM conference on Computer and communications security. ACM, 2003. pp. 290-299.
14. Крис К. Искусство дизассемблирования. СПб.: БХВ-Петербург. 2008, 896 с.

15. Алексеев В. Е., Таланов В. А. Графы и алгоритмы. Структуры данных. Модели вычислений. М.: Интернет-университет информационных технологий, 2006, 67 с.

REFERENCES:

[1] Aleinikov S. I., Bogatov A. Protecting programs against disassembling. Proceedings of Institute for system programming of the Russian Academy of Sciences. 2006. T. 11. Pp. 9-16. (in Russian).
[2] Collberg C., Thomborson C. and Low D. A Taxonomy of Obfucating Transformations,Technical Report N148, Department of Computer Science. The University of Auckland, July, 1997.
[3] Hlestov A. D. Nicholas K. Y., Obfuscation and protection methods software products. Vestnik UrFO. Security in the information sphere. 2015. No. 2. P. 7-10. (in Russian).
[4] Yarmolik V.N., Portyanko S.S. State of the ART in Software Ownership Protection. Computer Information Systems and Industrial Management Application Editors: Khalid Saeed, Romuald Mosdorf, Olli-Pekka Hilmola. Bialystok, Poland. 2003. pp. 188-195.
[5] Collberg C.S., Thomborson C. Watermarking, Tamper-proofing and Obfuscation – Tools for Software protection. Computer Science Technical Report #170. University of Auckland, Aucland, New Zealand. February 10, 2000.
[6] Kozubenko Y. V. copyright Protection for computer programs in the criminal, administrative and civil proceedings. Wolters Kluwer. Russia, 2009, 344 p. (in Russian).
[7] Alexander G. Gounares Fingerprinting Executable Code. Patent US20120317421 A1, date 19.07.2012.
[8] Anckaert B. et al. Steganography for executables and code transformation signatures. International Conference on Information Security and Cryptology. Springer Berlin Heidelberg. 2004. pp. 425-439.
[9] Zaidan A. A. et al. Novel approach for high (secure and rate) data hidden within triplex space for executable file. Scientific Research and Essays. 2010. vol. 5. no. 15. pp. 1965-1977.
[10] Collberg C., Thomborson C. Software watermarking: Models and dynamic embeddings. Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM. 1999. pp. 311-324.
[11] Moser A., Kruegel C., Kirda E. Limits of static analysis for malware detection. Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual. IEEE. 2007. pp. 421-430.
[12] Ernst M. D. Static and dynamic analysis: Synergy and duality. WODA 2003: ICSE Workshop on Dynamic Analysis. 2003. pp. 24-27.
[13] Linn C., Debray S. Obfuscation of executable code to improve resistance to static disassembly. Proceedings of the 10th ACM conference on Computer and communications security. ACM, 2003. pp. 290-299.
[14] Chris K. the Art of disassembly. SPb.: BHV-Petersburg. 2008, 896 p. (in Russian).
[15] Alekseev V. E., Talanov V. A. Graphs and algorithms. The structure of the data. The model calculations. M.: Internet-University of information technologies, 2006, 67 p. (in Russian).

Поступила в редакцию – 20 февраля июля 2017 г. Окончательный вариант – 21 мая 2017 г.
Received – February 20, 2017. The final version – May 21, 2017.