MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 2, 2020
journal homepage: www.mais-journal.ru

COMPUTER SYSTEM ORGANIZATION

On the Detection of Exploitation of Vulnerabilities Leading to the Execution of a Malicious Code

Y. V. Kosolapov[1]

DOI: 10.18255/1818-1015-2020-2-138-151

[1]Southern Federal University, 8a Milchakova str., Rostov-on-Don 344090, Russia.

MSC2020: 68M25
Research article
Full text in Russian

Received March 9, 2019
After revision March 23, 2020
Accepted March 25, 2020

Software protection from exploitation of possible unknown vulnerabilities can be performed both by searching (for example, using symbolic execution) and subsequent elimination of the vulnerabilities and by using detection and / or intrusion prevention systems. In the latter case, this problem is usually solved by forming a profile of a normal behavior and deviation from normal behavior over a predetermined threshold is regarded as an anomaly or an attack. In this paper, the task is to protect a given software P from exploiting unknown vulnerabilities. For this aim a method is proposed for constructing a profile of the normal execution of the program P, in which, in addition to a set of legal chains of system and library functions, it is proposed to take into account the distances between adjacent function calls. At the same time, a profile is formed for each program. It is assumed that taking into account the distances between function calls will reveal shell code execution using system and / or library function calls. An algorithm and a system for detecting abnormal code execution are proposed. The work carried out experiments in the case when P is the FireFox browser. During the experiments the possibility of applying the developed algorithm to identify abnormal behavior when launching publicly available exploits was investigated.

Keywords: system calls; library calls; software vulnerability.

INFORMATION ABOUT THE AUTHORS

Yury V. Kosolapov
correspondence author
orcid.org/0000-0002-1491-524X. E-mail: itaim@mail.ru
PhD.

For citation: Y. V. Kosolapov, “On the Detection of Exploitation of Vulnerabilities Leading to the Execution of a Malicious Code”, *Modeling and analysis of information systems*, vol. 27, no. 2, pp. 138-151, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 2, 2020
сайт журнала: www.mais-journal.ru

COMPUTER SYSTEM ORGANIZATION

Об обнаружении эксплуатации уязвимостей, приводящей к запуску вредоносного кода

Ю. В. Косолапов[1]

DOI: 10.18255/1818-1015-2020-2-138-151

[1]Южный Федеральный Университет, ул. Мильчакова, 8а, г. Ростов-на-Дону, 344090 Россия.

УДК 517.9
Научная статья
Полный текст на русском языке

Получена 9 марта 2019 г.
После доработки 23 марта 2020 г.
Принята к публикации 25 марта 2020 г.

Задача защиты программного обеспечения от эксплуатации возможных неизвестных уязвимостей может решаться как путем поиска (например, с помощью символьного исполнения) и последующего устранения уязвимостей, так и путем использования систем обнаружения и/или предотвращения вторжений. В последнем случае эта задача решается обычно путем формирования профиля нормального выполнения программ, а недопустимое отклонение от нормального состояния расценивается как аномалия или атака. В настоящей работе рассматривается задача защиты заданного исполнимого файла (программы) P от эксплуатации неизвестных уязвимостей в нем. Для этого предлагается способ построения профиля нормального выполнения программы P, в котором кроме набора легальных цепочек системных и библиотечных функций длины l учитывается расстояние между соседними вызовами функций, вычисляемое как разность адресов вызова соответствующих функций. Учет расстояний между вызовами функций позволяет выявлять исполнение вредоносного шеллкода, использующего вызовы системных и/или библиотечных функций, если хотя бы один из используемых в шеллкоде вызовов находится на нетипичном для программы P расстоянии от предыдущего вызова. В работе строится алгоритм и система обнаружения аномального выполнения кода и проводятся эксперименты в случае, когда P — браузер FireFox для операционной системы Windows.

Ключевые слова: системные вызовы; вызовы библиотек; уязвимости программного обеспечения.

ИНФОРМАЦИЯ ОБ АВТОРАХ

Юрий Владимирович Косолапов
автор для корреспонденции
orcid.org/0000-0002-1491-524X. E-mail: itaim@mail.ru
канд. техн. наук.

Для цитирования: Y. V. Kosolapov, “On the Detection of Exploitation of Vulnerabilities Leading to the Execution of a Malicious Code”, *Modeling and analysis of information systems*, vol. 27, no. 2, pp. 138-151, 2020.

Введение и постановка задачи

Выявление вредоносного программного обеспечения (ВПО) на основе аномального выполнения обладает преимуществом перед выявлением ВПО на основе его сигнатуры: первые позволяют обнаружить новые, еще не изученные образцы ВПО. В системах на основе выявления аномального поведения выделяются наборы признаков, которые отличают поведение легитимной программы от поведения ВПО [1]. Признаки в таких наборах часто строятся на основе последовательности API-вызовов (API — application program interface) или графа вызовов функций, используемых в ПО [2]. Например, в [3] мониторинг API-вызовов операционной системы Windows (функций из библиотек `kernel32.dll`, `advapi32.dll`, `gdi32.dll`, `comct32.dll`, `user32.dll`, `shell32.dll`, `ntdll.dll`) используется для классификации *исполнимых файлов* ВПО на пять различных классов: Worm (червь), Trojan-Downloader (загрузчик), Trojan-Spy (шпионское ПО), Trojan-Dropper (инсталлятор) и Backdoor (бэкдор). Для каждого из этих классов на основе взаимодействия ВПО с сетью, реестром, системными сервисами и т.п., происходящего посредством вызовов соответствующих API-функций, в [3] находится признак класса. В основе построения признака, сравнения и классификации лежит техника нечеткого хеширования (fuzzy hashing). В отличие от [3], где решение о классе ВПО принимается во время исполнения ВПО (т.е. динамически), в [4] решение о наличии вредоносного исполнения принимается на основе статического анализа графа API-вызовов. Для получения информации о вызываемых системных функциях используется дизассемблер IDA Pro. Заметим, что статический анализ затруднен для обфусцированных файлов. В [4] на основе техники DCFS (Document Class Frequency Selection) выбирают комбинации (n-граммы) API-функций, которые характерны для ВПО, а также комбинации API-функций, которые характерны для легитимных программ. На основе двух этих наборов строится вектор признаков (характеристик) для обучения SVM-классификатора. Статический анализ последовательности API-вызовов применен также и в [5], где классификатор строится на основе вычисления показателя похожести графов вызовов. В работе [6] для выявления ВПО и определения класса ВПО используются методы процессной аналитики (process mining): выполнение исполнимого файла описывается с помощью декларативной модели — конечного автомата, в котором переходы между состояниями помечены системными вызовами. Алгоритм выявления основан на вычислении расстояния между моделью, построенной для исследуемого исполнимого файла, и моделями известных семейств ВПО, и далее применении одного из алгоритмов классификации. Подробную классификацию систем обнаружения аномалий на основе системных вызовов можно найти, например, в [7]. Кроме признаков, строящихся исключительно на коде ВПО или на основе генерируемой последовательности машинных команд/API-вызовов, могут использоваться признаки, основанные на том, как ВПО преобразует данные. Например, для вирусов-шифровальщиков признаком выполнения может являться энтропия записываемых ими данных, которая для зашифрованных данных обычно выше энтропии незашифрованных данных.

Выявление аномального поведения может использоваться не только для обнаружения ВПО, запущенного в системе в виде отдельного исполнимого файла, но и для обнаружения эксплоитов, нарушающих работу легитимного ПО. Однако выявление исполнения кода, эксплуатирующего уязвимость, намного сложнее, чем выявление вредоносного кода, на который обычно передается управление эксплоитом [8]. Для выявления эксплоитов также необходимо иметь совокупность признаков, и «нормальные» значения этих признаков, которые бы характеризовали выполнение программы в соответствии с заложенными в нее алгоритмами. Отклонение значений от «нормальных» может свидетельствовать как о запуске эксплоита (истинное обнаружение), так и о том, что программа получила на вход легитимные данные, которые не были использованы на этапе обучения (ложное обнаружение). Шеллкод, на который передается управление эксплоитом после успешной эксплуатации уязвимости (например, при переполнении буфера в стеке или в куче), часто вызывает API-функцию, например, функцию выделения памяти `VirtualAlloc` в операци-

онной системе Windows. В общем случае, вызываемая функция может быть любой, в том числе и библиотечной, так как различные техники повторного использования исполнимого кода позволяют вызвать практически любую такую функцию (описание техник повторного использования исполнимого кода можно найти, например, в работах [9—11]). Так как выполнение эксплоита может привести к нарушению обычного порядка вызова системных и библиотечных функций, то признаком аномального выполнения легитимного ПО может являться нетипичная последовательность системных и библиотечных функций. Этот подход обнаружения аномалий подробно описан в [2], однако он может быть нивелирован в ряде случаев мимикрирующими атаками [12].

Так как после завершения загрузки статических библиотек адреса функций не меняются, то признаком аномального поведения легитимного ПО может быть расстояние между соседними вызовами функций в трассе вызовов, полученной после запуска программы (под расстоянием понимается разность адресов вызова функций). Сформированный для существующей в легитимном ПО уязвимости шеллкод, который вызывает системную или библиотечную функцию, может находиться на нетипичном для нормального выполнения расстоянии от вызова предыдущей или до вызова следующей отслеживаемой функции. Это относится и к шеллкоду на основе повторного использования исполнимого кода, и к шеллкоду, записываемому в динамически выделенную *исполнимую* память. В настоящей работе ставится задача оценки возможности применения расстояния между API-вызовами в качестве признака для обнаружения факта запуска эксплоита в легитимной программе. В первом разделе описывается способ построения профиля программы и алгоритм обнаружения, а во втором – описаны результаты экспериментов, проведенных для браузера FireFox.

1. Модель обнаружения аномального поведения

В настоящей работе, с одной стороны, предлагается по трассе системных и библиотечных вызовов строить цепочки вызовов, как описано в [2], а с другой стороны – с целью противодействия атакам маскировки под легальные цепочки вызовов, описанным в [12], учитывать расстояние между соседними системными вызовами и/или библиотечными вызовами. При внедрении нелегитимного кода (в том числе, с помощью техник повторного использования исполнимого кода [9—11]) появляется точка вызова функции (системной или библиотечной), расстояние от которой (или до которой) может быть *нетипичным*. Наличие нетипичного расстояния может служить признаком заражения и использоваться для защиты конкретных приложений (профиль типичных расстояний индивидуален для каждой программы). Защищаемыми приложениями могут быть, например, просмоторщики pdf-файлов, редакторы документов, браузеры.

1.1. Алгоритмы выявления аномального поведения

Пусть $\mathcal{L}(P)$ – множество имен библиотечных функций и функций ядра, используемых программой P. Заметим, что при запуске исполнимого файла (программы P) в виртуальную память процесса загружаются используемые им библиотеки. В случае, когда задействована технология рандомизации адресного пространства (Address Space Layout Randomization — ASLR), в разные моменты запуска исполнимого файла одна и та же библиотека может быть загружена по разным адресам. Отметим, что технология ASLR направлена на защиту от атак типа повторного использования исполнимого кода, однако такая защита может быть нивелирована, если в исполняемой программе удается найти и использовать уязвимость типа раскрытия памяти, при эксплуатации которой атакующий получает информацию об адресах загрузки библиотек [13]. Введем для удобства вспомогательную функцию B(`addr`), которая по адресу `addr` возвращает адрес модуля, адресному пространству которого этот адрес принадлежит. Пусть $\mathrm{Path}_t(P(I))$ — последовательность системных вызовов и вызовов библиотек *одного потока* программы P, загруженной в память в момент времени t и запускаемой с входными данными I:

$$\mathrm{Path}_t(P(I)) = (f_1^{t,I}, ..., f_{N_{t,I}}^{t,I}), \tag{1}$$

где API-вызов $f_k^{t,I}$ с порядковым номером k представим в виде тройки:

$$f_k^{t,I} = (B(\mathtt{ca}_k^{t,I}), \mathtt{ca}_k^{t,I}, n_k^{t,I}), \tag{2}$$

$B(\mathtt{ca}_k^{t,I})$ — адрес, по которому загружен исполнимый модуль (библиотека) в виртуальную память, содержащий по адресу $\mathtt{ca}_k^{t,I}$ вызов функции с именем $n_k^{t,I} (\in \mathcal{L}(P))$. Для $t_1 \neq t_2$ последовательности API-вызовов $\mathrm{Path}_{t_1}(P(I))$ и $\mathrm{Path}_{t_2}(P(I))$ (при одних и тех же входных параметрах программы) в общем случае, в силу ASLR, отличаются адресами $B(\mathtt{ca})$ и $\mathtt{ca}$ (см. таблицу 1).

Table 1. An example of a single program execution trace at times t_1 and t_2

Таблица 1. Пример трассы выполнения одной программы в моменты времени t_1 и t_2

N	$\mathrm{Path}_{t_1}(P(I))$	$\mathrm{Path}_{t_2}(P(I))$
1	$(B(\mathtt{ca}_1^{t_1,I}), \mathtt{ca}_1^{t_1,I}, n_1^{t_1,I})$	$(B(\mathtt{ca}_1^{t_2,I}), \mathtt{ca}_1^{t_2,I}, n_1^{t_2,I})$
2	$(B(\mathtt{ca}_2^{t_1,I}), \mathtt{ca}_2^{t_1,I}, n_2^{t_1,I})$	$(B(\mathtt{ca}_2^{t_2,I}), \mathtt{ca}_2^{t_2,I}, n_1^{t_2,I})$
3	$(B(\mathtt{ca}_3^{t_1,I}), \mathtt{ca}_3^{t_1,I}, n_3^{t_1,I})$	$(B(\mathtt{ca}_3^{t_2,I}), \mathtt{ca}_3^{t_2,I}, n_1^{t_2,I})$
⋮	⋮	⋮

Вообще говоря, $n_k^{t_1,I} \neq n_k^{t_2,I}$ в общем случае, при одних и тех же входных данных I (когда, исполнение программы зависит не только от входных данных, но и, например, от системного счетчика времени или значения на выходе генератора псевдослучайных чисел). Однако без потери общности, предполагается, что в таблице 1 вызовы с порядковым номером k (столбец N), как в момент времени t_1, так и в момент времени t_2 выполняются из одного и того же модуля (другими словами, по адресам $B(\mathtt{ca}_k^{t_1,I})$ и $B(\mathtt{ca}_k^{t_2,I})$ загружен один и тот же модуль, причем при разных k модули могут быть разными). Отсюда получаем, что в общем случае $d_{k,k+1}^{t_1,I} = \mathtt{ca}_{k+1}^{t_1,I} - \mathtt{ca}_k^{t_1,I} \neq \mathtt{ca}_{k+1}^{t_2,I} - \mathtt{ca}_k^{t_2,I} = d_{k,k+1}^{t_2,I}$ для $n_k^{t_1,I} = n_k^{t_2,I}$ и $n_{k+1}^{t_1,I} = n_{k+1}^{t_2,I}$. С другой стороны, для любого k при $n_k^{t_1,I} = n_k^{t_2,I}$ выполняется равенство $\mathtt{ca}_k^{t_1,I} - B(\mathtt{ca}_k^{t_1,I}) = \mathtt{ca}_k^{t_2,I} - B(\mathtt{ca}_k^{t_2,I})$. Тогда

$$\mathtt{ca}_k^{t_2,I} = \mathtt{ca}_k^{t_1,I} + (B(\mathtt{ca}_k^{t_2,I}) - B(\mathtt{ca}_k^{t_1,I})) = \mathtt{ca}_k^{t_1,I} + \Delta_k^{t_1,t_2,I} = M_{t_2}(\mathtt{ca}_k^{t_1,I}). \tag{3}$$

Поэтому,

$$d_{k,k+1}^{t_2,I} = M_{t_2}(\mathtt{ca}_{k+1}^{t_1,I}) - M_{t_2}(\mathtt{ca}_k^{t_1,I}) = d_{k,k+1}^{t_1,I} + \Delta_{k+1}^{t_1,t_2,I} - \Delta_k^{t_1,t_2,I}. \tag{4}$$

Таким образом, если накопление информации (обучение) о расстояниях между вызовами проводится в момент времени t_1, а проверка – в момент времени t_2, то выражение (3) позволяет спроецировать наблюдаемые в момент тестирования значения адресов на те адреса загрузки библиотек, которые были в момент обучения. Для $(n, m) \in \mathcal{L}(P) \times \mathcal{L}(P)$ будем писать $(n, m) \propto_d \mathrm{Path}_t(P(I))$, если найдется такое k, что $n = n_k^{t,I}$, $m = n_{k+1}^{t,I}$ и $d = \mathtt{ca}_{k+1}^{t,I} - \mathtt{ca}_k^{t,I}$ для троек $(B(\mathtt{ca}_k^{t,I}), \mathtt{ca}_k^{t,I}, n_k^{t,I})$ и $(B(\mathtt{ca}_{k+1}^{t,I}), \mathtt{ca}_{k+1}^{t,I}, n_{k+1}^{t,I})$. Пусть $\mathcal{I}(P)$ — множество значений *легитимных* входных данных программы P (множество тех данных, на которых строится профиль нормального поведения программы). Для каждой пары $(n, m) \in \mathcal{L}(P) \times \mathcal{L}(P)$ рассмотрим множество

$$D_{(n,m)}^t = \{d \in \mathbb{Z} | \exists I \in \mathcal{I}(P) \,:\, (n, m) \propto_d \mathrm{Path}_t(P(I))\}.$$

Другими словами, $D_{(n,m)}^t$ — множество возможных расстояний между API-функциями n и m, вызываемыми друг за другом (функция m вызывается после функции n). Отметим, что в общем случае $D_{(n,m)}^t \neq D_{(m,n)}^t$. *Профилем расстояний* программы P назовем множество

$$\mathcal{D}^t(P) = \left\{ D_{(n,m)}^t \,:\, (n, m) \in \mathcal{L}(P) \times \mathcal{L}(P) \right\}, \tag{5}$$

построенное *по всем потокам* программы. Пусть $l^t_{min}(P)$ и $l^t_{max}(P)$ — минимальная и максимальная длина списка расстояний в профиле $\mathcal{D}^t(P)$. Максимальное и минимальное (по профилю (5)) значение расстояния между соседними вызовами для программы P в момент запуска t_1 обозначим соответственно $d^{t_1}_{max}(P)$ и $d^{t_1}_{min}(P)$. Очевидно, что если спроецированное наблюдаемое значение расстояния между соседними API-вызовами не принадлежит диапазону

$$[d^{t_1}_{min}(P) \,:\, d^{t_1}_{max}(P)] = \{d^{t_1}_{min}(P), d^{t_1}_{min}(P)+1, ..., d^{t_1}_{max}(P)\},$$

то это свидетельствует об аномальном выполнении. *Профилем цепочек* порядка l программы P, в соответствии с [2], назовем множество

$$C_t(P,l) = \left\{(n_1,...,n_l) | \exists I \in \mathcal{I}(P) \exists k \in \mathbb{N} \forall r \in \{1,...,l\} \,:\, n^{t,I}_{k+r} = n_r\right\}. \tag{6}$$

Алгоритм *CheckTrace* (см. алгоритм 1) выявляет нетипичное выполнение программы P на основе профиля расстояний (5) и профиля цепочек (6).

Алгоритм 1. *CheckTrace*

Исходные параметры: 1) последовательность $\mathrm{Path}_{t_2}(P(I))$ вида (1) длины n_I ($I \notin \mathcal{I}(P)$),
2) профиль $\mathcal{D}_{t_1}(P)$ вида (5) и профиль $\mathcal{C}_{t_1}(P,l)$ вида (6)

Результат : Сообщение о нетипичной (`not typical`) или типичной (`typical`) последовательности API-вызовов

```
result = typical
цикл k = 1, ..., n_I выполнять
    если k ⩾ l и (n^{t2,I}_{k-l+1}, ..., n^{t2,I}_k) ∉ C_{t1}(P, l) тогда
        result = not typical
        Выйти из цикла
    если k ⩽ n_I − 1 тогда
        d = d^{t2,I}_{k,k+1} − Δ^{t1,t2,I}_{k+1} + Δ^{t1,t2,I}_k
        если d ∈ [d^{t1}_min(P) : d^{t1}_max(P)] тогда
            если d ∉ D_{(f^{t2,I}_k, f^{t2,I}_{k+1})} тогда
                result = not typical
        иначе
            result = not typical
            Выйти из цикла
возвратить result
```

В алгоритме сначала выполняется проверка, является ли текущая цепочка вызовов длины l из трассы вызовов $\mathrm{Path}_{t_2}(P(I))$ типичной, то есть содержится ли эта цепочка в профиле $\mathcal{C}_{t_1}(P,l)$. Отсутствие цепочки свидетельствует об отклонении от нормального исполнения. В случае, если цепочка присутствует в профиле, то проверяется является ли типичным расстояние между соседними API-вызовами. Отметим, в случае $l = 2$, проверка на типичность цепочки является избыточной, так как в этом случае $D_{n,m} = \varnothing$ для нетипичной пары (n, m), и достаточно проверки по профилю расстояний.

Утверждение 1. *Пусть* $[A_1 \,:\, A_2]$ *— диапазон адресов виртуальной памяти в момент времени* t_1*, используемый для загрузки кода программы* P *и требуемых модулей/библиотек,* $[E_1 \,:\, E_2]$ *— диапазон*

адресов памяти в момент времени t_2 отслеживаемого API-вызова из шеллкода, содержащегося во входных данных I, $[A_1 : A_2] \cap [M_{t_2}^{-1}(E_1) : M_{t_2}^{-1}(E_2)] = \varnothing$. Если $A_2 - A_1 < M_{t_2}^{-1}(E_1) - A_2$ либо $A_2 - A_1 < A_1 - M_{t_2}^{-1}(E_2)$, то

$$CheckTrace(\mathrm{Path}_{t_2}(P(I)), \boldsymbol{D}_{t_1}(P), C_{t_1}(P, l), 1) = \texttt{not typical}$$

для любого $t_2 \geqslant t_1$.

Доказательство. Доказательство утверждения вытекает из того, что расстояние между легитимными вызовами будет не более $A_2 - A_1$, а шеллкод обращается к API-функции с адреса, который находится от предшествующего API-вызова на расстоянии d, большем $A_2 - A_1$. Следовательно, число d не будет в списке возможных расстояний и алгоритм $CheckTrace$ вернет результат проверки `not typical`. Этот результат будет возвращен в любой момент времени $t_2 \geqslant t_1$, так как выражение (4) позволяет перейти от адресации в момент времени t_2 к адресации в момент t_1. □

Отметим, что в операционной системе Windows диапазон адресов загрузки модулей библиотек и диапазон области кучи, в которой часто размещается шеллкод, пересекаются, поэтому условия утверждения 1 для этой операционной системы обычно не выполняются. Следующие два утверждения доказываются по аналогии с утверждением 1.

Утверждение 2. *Пусть $[A_1 : A_2]$ — диапазон адресов виртуальной памяти в момент времени t_1, из области которого отслеживаются API-вызовы программы P, $[E_1 : E_2]$ — диапазон адресов памяти в момент времени t_2 отслеживаемого API-вызова из шеллкода, содержащегося во входных данных I, $[A_1 : A_2] \cap [M_{t_2}^{-1}(E_1) : M_{t_2}^{-1}(E_2)] = \varnothing$. Если $A_2 - A_1 < M_{t_2}^{-1}(E_1) - A_2$ либо $A_2 - A_1 < A_1 - M_{t_2}^{-1}(E_2)$, то*

$$CheckTrace(\mathrm{Path}_{t_2}(P(I)), \boldsymbol{D}_{t_1}(P), C_{t_1}(P, l), 1) = \texttt{not typical}$$

для любого $t_2 \geqslant t_1$.

Утверждение 3. *Пусть $[A_1 : A_2]$ — диапазон адресов памяти в момент времени t_1, используемый для загрузки кода программы P и требуемых модулей/библиотек, D — максимальное расстояние между отслеживаемыми API-вызовами программы P в момент времени t_1 для всех легитимных входных данных, $[E_1 : E_2]$ — диапазон адресов памяти в момент времени t_2 отслеживаемого API-вызова из шеллкода, содержащегося во входных данных I, $[A_1 : A_2] \cap [M_{t_2}^{-1}(E_1) : M_{t_2}^{-1}(E_2)] = \varnothing$. Если $D < M_{t_2}^{-1}(E_1) - A_2$ либо $D < A_1 - M_{t_2}^{-1}(E_2)$, то*

$$CheckTrace(\mathrm{Path}_{t_2}(P(I)), \boldsymbol{D}_{t_1}(P), C_{t_1}(P, l), 1) = \texttt{not typical}$$

для любого $t_2 \geqslant t_1$.

Пусть $\boldsymbol{I}(P)$ и $\boldsymbol{I}^E(P)$ — множество легитимных и нелегитимных входных значений программы P, $\boldsymbol{I}^E(P) \cap \boldsymbol{I}(P) = \varnothing$, $N^L = |\boldsymbol{I}(P)|$, $N^E = |\boldsymbol{I}^E(P)|$, A – алгоритм обнаружения, использующий профиль $\boldsymbol{D}_{t_1}(P)$ вида (5) и профиль $C_{t_1}(P, l)$ вида (6),

$$J^A(I) = \begin{cases} 1, & A(\mathrm{Path}_{t_2}(P(I))) = \texttt{not typical}, \\ 0, & A(\mathrm{Path}_{t_2}(P(I))) = \texttt{typical}. \end{cases}$$

Важными характеристиками систем обнаружения аномалий являются: вероятность ложного обнаружения P_{FP}^A, вероятность истинного обнаружения P_{TP}^A (чувствительность обнаружения или уровень обнаружения) [1]:

$$P_{FP}^A = \frac{\sum_{I \in \boldsymbol{I}(P)} J^A(I)}{N^L}, P_{TP}^A = \frac{\sum_{I \in \boldsymbol{I}^E(P)} J^A(I)}{N^E}.$$

Для возможного снижения вероятности ложного обнаружения разработан также алгоритм обнаружения $CheckTraceBack$ (см. алгоритм 2), в котором учитывается расстояние не до одного предыдущего вызова, а расстояния до $T+1$ предыдущих вызовов. Именно, в случае, когда обнаруживается нетипичное расстояние между соседними вызовами (в случае, когда алгоритм $CheckTrace$ возвращает результат `not typical`), предлагается вычислять двоичный вектор вида

$$\mathbf{b}_{f_k^{t_2,I},f_{k+1}^{t_2,I}} = (b_1, ..., b_T) \in \{0,1\}^T, \tag{7}$$

i-ая координата ($i \in \{1, ..., T\}$) которого определяется по правилу:

$$b_i = \begin{cases} 1, d_{k-i,k+1}^{t_2,I} - \Delta_{k+1}^{t_1,t_2,I} + \Delta_{k-i}^{t_1,t_2,I} \notin D_{f_{k-i}^{t_2,I},f_{k+1}^{t_2,I}} \\ 0, \text{ иначе} \end{cases}.$$

В алгоритме $CheckTraceBack$ аномальным поведением считается вызов $f_{k+1}^{t_2,I}$, для которого расстояние до предыдущего вызова является нетипичным и вес Хэмминга вектора (7) равен T: все расстояния до вызова $f_{k+1}^{t_2,I}$ от предыдущих $T+1$ вызовов являются нетипичными (что представляется характерным при выполнении эксплоита). Отметим, что в общем случае можно ввести порог на вес вектора (7), при превышении которого вызов считается нетипичным. Однако это обобщение в настоящей работе не рассматривается.

Отметим ряд ограничений разработанных алгоритмов. Во-первых, отслеживание API-вызовов для определенного диапазона адресов (как в утверждении 2) может привести к пропуску запуска эксплоита, даже если API-функция является отслеживаемой (например, если вызов API-функции выполнен из библиотеки, вызовы из которой не отслеживаются). Во-вторых, если эксплоит использует функции, не входящие в $\mathcal{L}(P)$, либо вообще не использует API-вызовы, то его выполнение не будет обнаружено (целью таких эксплоитов обычно является остановка работы приложения). В-третьих, предлагаемый способ защиты не обнаруживает запуск вредоносного кода, если эксплуатация уязвимости не предполагает внедрение/запись исполнимого кода в память уязвимой программы. В частности, атака на основе динамического обмена данными (Dynamic Data Exchange — DDE), описанная в [14], не будет выявлена, так как в этом случае нормальное исполнение программы неотличимо от аномального только на основе вызываемых API-функций и расстояний между ними. Для выявления таких атак необходим анализ входных параметров вызываемых функций, что усложняет алгоритм обнаружения.

Создатель вредоносного кода может попытаться замаскировать выполнение своего кода под легальный профиль: для маскировки необходимо, чтобы адрес инструкции, вызывающей API-функцию, был на легитимном расстоянии от предыдущей вызванной API-функции и на легитимном расстоянии до следующей API-функции, а также чтобы результирующая последовательность API-вызовов находилась в базе легитимных цепочек. Если информация об адресах загрузки модулей в момент времени t_1 (в момент обучения) доступна создателю вредоносного кода, то он может, используя подходящую уязвимость, в результате эксплуатации которой в момент времени t_2 (в момент эксплуатации) происходит утечка информации о базовых адресах загруженных модулей, подготовить эксплоит, в котором вызов API-функции будет находиться на легитимном расстоянии (вычисленном в соответствии с (4)) от предыдущего API-вызова и на легитимном расстоянии до следующего API-вызова. Подбор подходящего расстояния может быть выполнен с помощью вставки команд `nop` в машинный код. Поэтому информация об адресах загрузки модулей в момент t_1 должна быть доступна только системе защиты. При удаленной эксплуатации эта информация, как правило, неизвестна создателю кода. При локальной эксплуатации эту информацию можно сделать труднодоступной, например, храня ее в зашифрованном файле и расшифровывая только в момент загрузки системы защиты (ключ расшифрования, например, может храниться в исполнимом файле

Алгоритм 2. *CheckTraceBack*

Исходные параметры: 1) последовательность $\mathrm{Path}_{t_2}(P(I))$ вида (1) длины n_I ($I \notin \mathcal{I}(P)$), 2) профиль $\mathcal{D}_{t_1}(P)$ вида (5) и профиль $\mathcal{C}_{t_1}(P, l)$ вида (6), порог обнаружения T

Результат : Сообщение о нетипичной (`not typical`) или типичной (`typical`) последовательности API-вызовов

```
result = typical
цикл k = 1, ..., n_I выполнять
    если k ⩾ l и (n^{t2,I}_{k−l+1}, ..., n^{t2,I}_k) ∉ C_{t1}(P, l) тогда
        result = not typical
        Выйти из цикла
    если k ⩽ n_I − 1 тогда
        d = d^{t2,I}_{k,k+1} − Δ^{t1,t2,I}_{k+1} + Δ^{t1,t2,I}_k
        если d ∈ [d^{t1}_min(P) : d^{t1}_max(P)] тогда
            если d ∉ D_{f^{t2,I}_k, f^{t2,I}_{k+1}} тогда
                result = not typical
        иначе
            result = not typical
        если result = not typical тогда
            Вычислить вектор b вида (7)
            если wt(b) = T тогда
                Выйти из цикла
            иначе
                result = typical
                Добавить расстояние d в профиль D_{t1}(P) для пары (f^{t2,I}_k, f^{t2,I}_{k+1}).
возвратить result
```

системы защиты, защищенном подходящими методами обфускации [15]). Отметим, что даже в случае, когда информация об адресах загрузки модулей известна разработчику эксплоита, написание кода таким образом, чтобы вызываемые API-функции, с одной стороны, не приводили к появлению нетипичных цепочек (то есть чтобы цепочки принадлежали $\mathcal{C}_{t_1}(P, l)$), а с другой стороны, находились на легитимных расстояниях в типичных цепочках (то есть чтобы расстояния между вызовами принадлежали $\mathcal{D}_{t_1}(P)$), является сложной задачей. Также представляется, что список отслеживаемых API-функций должен оставаться в секрете: разработчику эксплоита для подбора расстояния необходима информация о том, какой API-вызов будет перед API-вызовом в шеллкоде.

1.2. Схема системы обнаружения аномального поведения

Схема системы обнаружения аномального поведения показана на рис. 1. Как и в большинстве систем выявления аномального поведения [1], в предлагаемой системе предусмотрен этап обучения, в ходе которого строятся профили $\mathcal{D}_t(P)$ и $\mathcal{C}_t(P, l)$ защищаемой программы. Входными данными для этого этапа являются набор $\mathcal{I}(P)$ (передается в блок сбора данных), набор $\mathcal{L}(P)$ (передается в блок настройки параметров) и натуральное число l (передается в блок построения профилей).

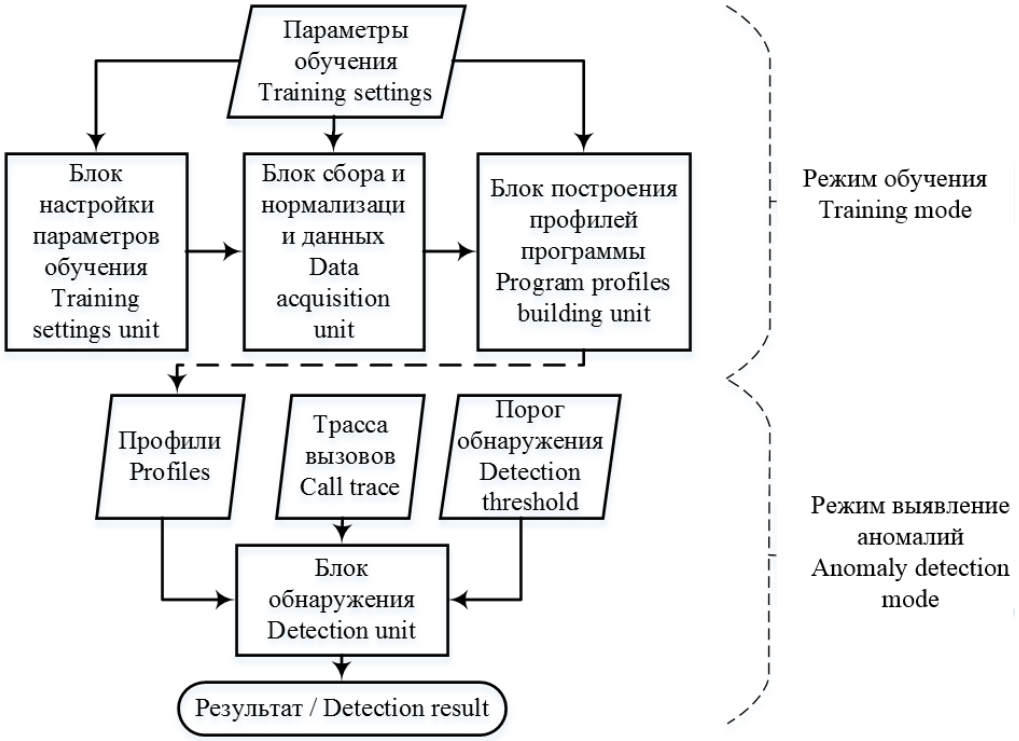

Fig. 1. Abnormal Behavior Detection System Diagram

Рис. 1. Схема системы обнаружения аномального поведения

Отметим, что чем больше мощность $\mathcal{I}(P)$, тем меньше вероятность принять нормальное поведение программы за аномалию. С другой стороны, увеличение $\mathcal{I}(P)$ приводит к увеличению периода обучения. В свою очередь увеличение $\mathcal{L}(P)$ может затруднить создателю эксплоита написание необнаруживаемого вредоносного кода. Однако увеличение числа отслеживаемых API-вызовов приводит к росту размера профиля и замедлению выполнения программы P. Актуальной задачей является выбор оптимальных наборов $\mathcal{I}(P)$ и $\mathcal{L}(P)$ (эта задача в настоящей работе не решается). Размер l цепочек в профиле $C_t(P, l)$ влияет на размер этого профиля. При малых значениях профиль компактный, но при этом возрастает вероятность пропуска вредоносного кода. При больших значениях l профиль растет и растет вероятность принять легитимную программу за вредоносную. В [2] экспериментальным путем установлено, что для операционных систем семейства Unix оптимальным значением l для обнаружения аномалий является 6.

Кратко опишем схему на рис. 1. В блоке сбора данных по выполняющейся программе P строится последовательность API-вызовов вида (1). Существуют различные способы мониторинга API-вызовов. Двумя наиболее распространенными способами является: 1) замена адресов импортируемых функций на адрес своего перехватчика и 2) замена начала вызываемых функций кодом перехода на свой обработчик [3]. В первом случае выполнение эксплоита может быть не обнаружено, так как эксплоит может вызывать функцию не через таблицу импорта, а передавать управление непосредственно на код самой API-функции. Поэтому для мониторинга следует использовать метод модификации начала импортируемых функций. Для мониторинга API-функций библиотек, загружаемых динамически, необходимо иметь возможность отслеживать загрузку библиотек. Средство мониторинга API-вызовов, реализующее блок сбора данных, должно позволять получать не только адрес ca API-вызова, но и адрес ba загрузки модуля, из которого выполняется этот вызов. Например, утилита API Monitoring [16] позволяет отслеживать для выбранного исполнимого файла

API-функции, вызываемые этим файлом, а также имеется возможность выбирать функции, вызов которых требуется отслеживать. В свою очередь средство ListDll [17] позволяет для активного процесса получить имена, адреса и размеры библиотек, загруженных в адресное пространство процесса. Таким образом, в совокупности эти средства позволяют построить последовательность троек вида (2), и поэтому может быть применено выражение (4). В блоке нормализации данные, накопленные в блоке сбора, проходят предварительную обработку. В частности, в этом блоке удаляются дубликаты API-вызовов. В блоке построения профиля формируются профили вида (5) и (6). Заметим, что для составления профиля (5) следует использовать трассы вызовов для разных потоков, и не использовать API-вызовы на границе переключения потоков, так как переключение потоков может быть в любой момент, и часто не зависит от логики программы. Построенные профили, вместе с трассой API-вызовов $\mathrm{Path}_{t_2}(P(I))$, $I \notin \mathcal{I}(P)$ затем используются как входные данные в блоке обнаружения, который реализует алгоритмы $CheckTrace$ (алгоритм 1) и $CheckTraceBack$ (алгоритм 2).

2. Эксперименты

Интернет-браузеры являются одними из наиболее часто используемых программ, что служит причиной большого внимания к этим программам со стороны создателей вредоносного кода и со стороны специалистов по защите [18, 19]. С целью исследования возможности применения разработанных алгоритмов в качестве защищаемой программы P выбран браузер FireFox. Для этого браузера в силу его популярности в открытом доступе имеются концептуальные эксплоиты [20—23], которые в настоящей работе используются в исследовательских целях.

Эксплоит, разработанный в работе [20], используя технику JIT-Spray, размещает исполнимый код (шеллкод) в памяти браузера FireFox версии 44.0.2 и передает на него управление при успешной эксплуатации уязвимости. В шеллкоде вызывается функция `WinExec` (из библиотеки `kernel32.dll`) для запуска калькулятора (приложение `calc.exe`). Отметим, что в приложении FireFox не предусмотрен вызов этой API-функции, поэтому функция `WinExec` не будет содержаться в профилях $\mathcal{D}_{t_1}(P)$ и $\mathcal{C}_{t_1}(P)$. Следовательно такой вызов будет расценен как аномалия в разработанных алгоритмах. Вообще говоря, вызов любой API-функции из шеллкода, не содержащейся в профилях $\mathcal{D}_{t_1}(P)$ и $\mathcal{C}_{t_1}(P)$, будет расценен как аномалия. К числу нетипичных API-вызовов для программы FireFox, кроме `WinExec`, относятся такие библиотечные API-функции как `ReadProcessMemory`, `WriteProcessMemory`, `CreateProcess` (отметим, что эти функции могут использоваться ВПО [3]). Поэтому интерес представляют шеллкоды, в которых вызываются API-функции, являющиеся типичными для трассы вызовов при легитимном выполнении приложения P. Как показано в [3], к числу типичных для ВПО функций относятся функции из таб. 2, которые, как также являются типичными и для программы FireFox. С этой целью в настоящей работе созданы модификации шеллкода из [20], в каждой из которых вызывается API-функция из таб. 2. Таким образом, $N^E = 15$. В качестве отслеживаемых API-функций, то есть тех функций, из вызовов которых формируется трасса (1), выбраны также все функции из таблицы 2. Входными данными для браузера FireFox выбраны сайты пятнадцати различных тематик в соответствии с классификацией, используемой на сайте `www.similarweb.com`: 1) новости, 2) сайты государственных учреждений, 3) искусство и развлечения, 4) бизнес, 5) сайты сообществ/социальных сетей, 6) технологии, 7) электронная коммерция, 8) финансовые сервисы, 9) продукты/еда, 10) игровые сайты, 11) хобби, 12) поиск работы 14) устройства/сенсоры, 15) путешествия. Список сайтов для каждой группы (тематики) формировался с помощью сервиса `www.similarweb.com`. Для исследования зависимости вероятности ложного обнаружения от размера обучающей выборки сформировано случайным образом 15 выборок размера от 10 до 150 сайтов с шагом 10. Для формирования каждой выборки использовались первые 10 наиболее посещаемых сайтов каждой группы[1]; каждая последующая выборка содержит предыдущую

[1] по состоянию на 20.02.2020

Table 2. Examples of the API functions typical for the FireFox and typical for the malware

Таблица 2. Примеры API-функций, типичных для программы FireFox и типичных для вредоносного ПО

Вредоносная активность/ Malicious activity	**API**	**DLL**
Клавиатурный шпион/ Key Logger	`FindWindowW/FindWindowExW`, `SetWindowsHookExA/SetWindowsHookExW`	`user32.dll`
Захват экрана/ Screen Capture	`GetDC`, `GetWindowDC`	`user32.dll`
	`CreateCompatibleDC`, `CreateCompatibleBitmap`	`gdi32.dll`
	`WriteFile/WriteFileEx`	`kernel32.dll`
Противодействие отладке/ Antidebugging	`IsDebuggerPresent`	
Внедрение кода/ DLL Injection	`VirtualAlloc/VirtualAllocEx`	
Инсталлятор/ Dropper	`LoadResource`, `SizeOfResource`	

(меньшего размера). В качестве тестовых наборов сформировано случайным образом 30 наборов по 15 сайтов в каждом наборе. Для тестовых наборов использовались первые 30 сайтов каждой тематики. Таким образом обучающая выборка и тестовая выборка пересекаются, что характеризует ситуацию, когда пользователь при обучении и при тестировании может посещать одинаковые сайты. Результаты оценки вероятности ложного обнаружения P_{FP}, усредненные по 30 тестовым выборкам, максимальная $l^{t_1}_{max}(P)$ и средняя $\mathrm{E}[l^{t_1}(P)]$ длина списка расстояний, а также размер S профиля расстояний (в байтах) приведены в таблице 3. Отметим, что при проведении экспериментов в алгоритмах $CheckTrace$ и $CheckTraceBack$ значение l выбрано равным двум, с целью оценить возможность обнаружения аномалий только на основе профиля расстояний. В качестве порога T выбраны значения 30 и 60.

Как видно из таблицы, с ростом размера обучающей выборки вероятность ложного обнаружения для обоих алгоритмов уменьшается. При этом для алгоритма $CheckTraceBack$ вероятность ложного обнаружения приближается к нулю уже при 150-ти сайтах в обучающей выборке. Из таблицы также видно, что резкое уменьшение вероятности ложного обнаружения при переходе от выборки размера 100 к выборке размера 110 сопровождается ростом размера базы (столбец S). Это может свидетельствовать о том, что на этапе обучения в выборке из 110 сайтов были такие сайты, которые в большей степени задействовали функционал программы P, чем сайты из предыдущих выборок. Для всех модификаций эксплоита оба алгоритма показали стопроцентное обнаружение при всех возможных значениях $|\mathcal{I}(P)|$.

3. Заключение

К недостаткам разработанного способ, кроме отмеченных в разделе 1.1, следует отнести необходимость запуска приложения, что может привести к запуску эксплоита. Тем не менее полагается, что позднее обнаружение эксплоита предпочтительнее его пропуска. Кроме того, алгоритмы могут быть адаптированы для динамического обнаружения эксплоитов на ранней стадии запуска, при выявлении нетипичного расстояния между функциями: если проверка нетипичности вызова осуществляется до выполнения вызываемой функции. Однако в этом случае возможно замедле-

Table 3. Dependence of the $CheckTrace$ and $CheckTraceBack$ algorithms characteristics on the training sample size for the FireFox

Таблица 3. Зависимость характеристик алгоритмов $CheckTrace$ и $CheckTraceBack$ от размера обучающей выборки для программы FireFox

$\lvert\mathcal{I}(P)\rvert$	P_{FP}				$l^{t_1}_{max}(P)$	$\mathrm{E}[l^{t_1}(P)]$	S
	$CheckTrace$		$CheckTraceBack$				
	$T = 30$	$T = 60$	$T = 30$	$T = 60$			
10	1	1	1	1	27	4	4634
20	1	1	0,38125	0,28	31	4	5499
30	1	1	0,35625	0,27	32	5	5742
40	1	1	0,3125	0,228	32	5	5910
50	1	0,946	0,3125	0,212	33	5	6010
60	1	0,826	0,3125	0,21	34	5	6140
70	0,99375	0,786	0,275	0,186	34	5	6254
80	0,84375	0,672	0,2375	0,164	35	5	6439
90	0,725	0,592	0,2375	0,162	37	6	6676
100	0,61875	0,506	0,2375	0,162	38	6	6977
110	0,36875	0,322	0,03125	0,012	38	6	7450
120	0,29375	0,294	0,03125	0,012	43	6	7658
130	0,2875	0,274	0,03125	0,01	44	6	7780
140	0,2375	0,218	0,03125	0,01	44	6	8095
150	0,21875	0,21	0,01875	0,004	45	6	8274

ние выполнения защищаемой программы P. К недостаткам следует отнести также необходимость переобучения системы защиты при обновлении защищаемого программного обеспечения.

Не смотря на отмеченные недостатки, рассмотренный подход на основе подсчета расстояний между соседними вызовами функций может использоваться при исследовании недоверенных входных данных на наличие в них эксплоита. Для оценки вероятности ложного пропуска P_{TP}, а также для уточнения оптимального значения T требуется проведение исследования для разных типов известных эксплоитов.

References

[1] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, “Survey of intrusion detection systems: techniques, datasets and challenges”, *Cybersecurity*, vol. 2, no. 1, p. 20, 2019.

[2] S. Forrest, S. Hofmeyr, and A. Somayaji, “The Evolution of System-Call Monitoring”, in *Proceedings of 2008 Annual Computer Security Applications Conference (ACSAC)*, 2008, pp. 418–430.

[3] S. Gupta, H. Sharma, and S. Kaur, “Malware Characterization Using Windows API Call Sequences”, *Journal of Cyber Security and Mobility*, vol. 7, no. 4, pp. 363–378, 2018.

[4] R. Veeramani and N. Rai, “Windows API based Malware Detection and Framework Analysis”, *International Journal of Scientific & Engineering Research*, vol. 3, no. 3, pp. 1–6, 2012.

[5] A. Singh, R. Arora, and H. Pareek, “Malware Analysis using Multiple API Sequence Mining Control Flow Graph”, *CoRR. arXiv preprint arXiv:1707.02691*, 2017.

[6] M. L. Bernardi, M. Cimitile, D. Distante, F. Martinelli, and F. Mercaldo, “Dynamic malware detection and phylogeny analysis using process mining”, *International Journal of Information Security*, vol. 18, no. 3, pp. 257–284, 2019.

[7] L. Viljanen, "A Survey of Application Level Intrusion Detection", *Technical report, Series of Publications C, Report C-2004-61 Helsinki*, 2004.

[8] G. Creech, "Developing a high-accuracy cross platform Host-Based Intrusion Detection System capable of reliably detecting zero-day attacks", PhD thesis, University of New South Wales, Canberra, Australia, 2014.

[9] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks", in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 969–986.

[10] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block Oriented Programming: Automating Data-Only Attacks", in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1868–1882.

[11] Y. V. Kosolapov, "About detection of code reuse attacks", *Modelirovanie i Analiz Informatsionnykh Sistem*, vol. 26, no. 2, pp. 213–228, 2019.

[12] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems", in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 255–264.

[13] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization", in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 574–588.

[14] E. Stalmans and S. El-Sherei, *Macro-less Code Exec in MSWord*, Last access 12.12.2019. [Online]. Available: https://sensepost.com/blog/2017/macro-less-code-exec-in-msword/.

[15] P. D. Borisov and Y. V. Kosolapov, "On the automatic analysis of the practical resistance of obfusting transformations", *Modelirovanie i Analiz Informatsionnykh Sistem*, vol. 26, no. 3, pp. 317–331, 2019.

[16] *API Monito*, Last access 28.11.2019. [Online]. Available: http://www.rohitab.com/apimonitor.

[17] *ListDLLs*, Last access 28.11.2019. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/downloads/listdlls.

[18] M. Vervier, M. Orru, B. J. Wever, and E. Sesterhenn, *Browser Security Whitepaper*, Last access 05.12.2019. [Online]. Available: https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf.

[19] R. Gawlik and T. Holz, "Sok: Make JIT-spray great again", in *WOOT'18 Proceedings of the 12th USENIX Conference on Offensive Technologies*, 2018, pp. 1–14.

[20] *Offensive Security, Exploitdb/exploits/windows/remote/42484.html*, Last access 05.12.2019. [Online]. Available: https://github.com/offensive-security/exploitdb/blob/master/exploits/windows/remote/42484.html.

[21] 0vercl0k, *CVE-2019-9810*, Last access 05.12.2019. [Online]. Available: https://github.com/0vercl0k/CVE-2019-9810.

[22] *Exploit Database*, Last access 05.12.2019. [Online]. Available: https://www.exploit-db.com/.

[23] *CVE-2017-5375_ASM.JS_JIT-Spray*, Last access 30.12.2019. [Online]. Available: https://github.com/rh0dev/expdev/tree/master.