*Д. Ю. Гамаюнов, А. А. Сковорода (Щербинина)*

# ОБНАРУЖЕНИЕ ПОЛИМОРФНОГО ШЕЛЛКОДА НА ОСНОВЕ ПОДОБИЯ

## Введение

Проблема вредоносного программного обеспечения (ВПО), появившаяся в конце прошлого столетия, со временем становится все более актуальной. Сложность разрабатываемых систем увеличивается, а вместе с ней расширяется и множество потенциальных ошибок в них, которые могут быть использованы ВПО для совершения атак. Развиваются средства обнаружения ВПО и защиты от него, и само ВПО развивается вместе с ними. Техники, используемые ВПО, становятся сложнее для обхода систем безопасности. Атаки внедрения кода (code injection attacks) являются одним из основных способов распространения вредоносного программного обеспечения. Такие атаки возможны за счет наличия уязвимостей, т. е. некоторых недостатков в системе, способных привести к ее неправильному функционированию (в частности, к нелегитимному использованию). Код, внедряемый в атакуемую программу, а затем исполняемый ею, называется шеллкодом [1]. Для обнаружения шеллкодов используется множество способов, большинство которых основано на выявлении некоторых отличительных характеристик вредоносного кода. Такими характеристиками могут служить:

• NOP-след — последовательность байтов, которая корректно исполняется процессором, начиная с произвольной позиции, передавая управление смысловой нагрузке шеллкода в результате своего исполнения.

Для обнаружения шеллкодов по NOP-следу используются различные методы, например частотный анализ [2] и метод абстрактной интерпретации [3].

• Дешифратор [4, 5, 6, 7] — участок кода, необходимый для расшифровки зашифрованной полезной нагрузки шеллкода перед исполнением.

Существуют методы обнаружения дешифраторов шеллкода в сетевом трафике, основывающиеся на характерной циклической структуре дешифратора [5]. Другие методы базируются на обязательном выполнении дешифратором самомодификаций [6, 7].

• GetPC-код [5].

В случае, если шеллкод использует шифрование, его дешифратору необходимо получить абсолютный адрес зашифрованной части. Для этого используется GetPC-код — код, который позволяет вычислить текущее состояние счетчика адреса.

• Зона адреса возврата.

Для корректного исполнения на целевой машине шеллкод, использующий уязвимость переполнения буфера, должен изменить зону адреса возврата эксплуатируемой программы, передав управление на себя (в NOP-след). Поэтому зона адреса возврата в таких шеллкодах представляет собой последовательность байтов (достаточно большую, порядка нескольких десятков байтов) из наиболее вероятных адресов NOP-зоны шеллкода после внедрения. Такая последовательность байтов является специфичной для конкретной эксплуатируемой программы. Buttercup [8] реализует метод нахождения зон адреса возврата в сетевом трафике, основываясь на знаниях о запущенных на защищаемой машине сервисах и возможных уязвимостях в них.

Перечисленные методы обнаружения используют эвристические методы, основанные на предположении, что вредоносный код обязательно содержит соответствующую характеристику. Однако шеллкод может не содержать ни одной из перечисленных особенностей — так называемый простой шеллкод (plain shellcode). Для простых шеллкодов методы обнаружения основаны на поиске некоторых последовательностей инструкций, для которых известно, что они

были встречены ранее в атаках. К таким методам относятся, например, сигнатурные методы обнаружения атак [9]. В связи с этим для атакующего имеет смысл менять вид шеллкода (последовательность байтов, которой он представляется), сохраняя его смысловую нагрузку. Для обнаружения таких шеллкодов сигнатурный анализ плохо подходит.

Действия, направленные на видоизменение и усложнение шеллкода, называются обфускацией шеллкода, а сами экземпляры шеллкодов, полученные таким образом, называются обфусцированными. Одним из способов обфускации является шифрование. Под полиморфными модификациями подразумеваются шеллкоды, полученные из одного и того же шеллкода различными обфусцирующими преобразованиями (действиями, применяемыми для видоизменения шеллкода). Работа направлена на обнаружение полиморфных модификаций шеллкодов из некоторого множества известных образцов.

## 1. Предлагаемый метод

В работе предложен алгоритм выявления полиморфных модификаций заданного образца. Он содержит несколько возможных вариантов анализа. Вот его основные шаги:

1. Определить, является ли исследуемый образец зашифрованным.

2. Если исследуемый образец зашифрован, получить расшифрованный образец, используя эмуляцию исполнения инструкций.

3. Очистить полученный образец от мусорных инструкций. Этот шаг используется не во всех вариантах анализа.

4. Установить подобие между полученным образцом и заданным примером шеллкода (из множества известных примеров шеллкодов) на основе сравнения, специфичного для используемого варианта анализа. На этом шаге образец сравнивается со всеми известными примерами шеллкодов.

Варианты анализа на 3-м и 4-м шагах:

• байтовый анализ;

• анализ на основе построения графа потока управления;

• анализ на основе построения трасс.

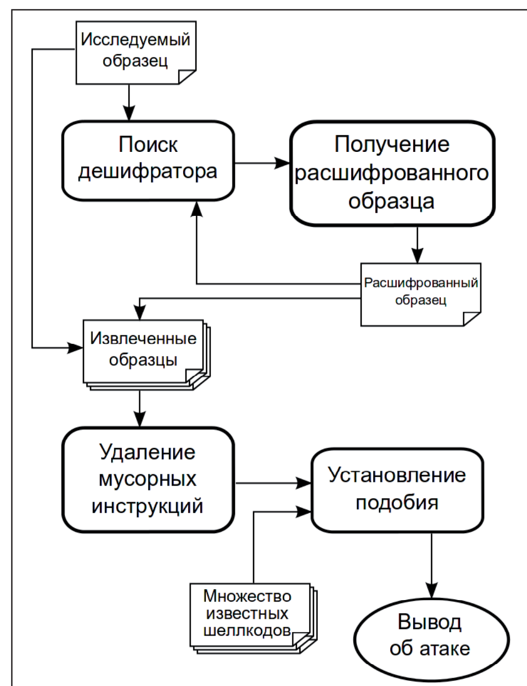


*Рис. 1. Схема работы алгоритма*

### 1.1. Определение зашифрованности

Для определения зашифрованности образца можно использовать различные способы.

Во-первых, можно предположить, что образец, использующий самомодификации, является зашифрованным. Ошибки, которые возникнут из-за такого отождествления понятий, не существенны, поскольку на последующем шаге они обязательно отразятся как небольшая длина расшифрованного образца (длина одной машинной инструкции), а мы можем ограничить ее снизу. Выявление же самомодифицируемости можно реализовать с помощью эмуляции исполнения инструкций, вычисляя для каждой инструкции записи в память адрес обращения и проверяя его на расположение в секции кода (в некоторой близости от текущего исполняемого адреса). Ограничение такого подхода — необходимо заранее определить адрес начала эмуляции. Это достаточно серьезная проблема, для ее решения можно использовать, например, перебор точек входа.

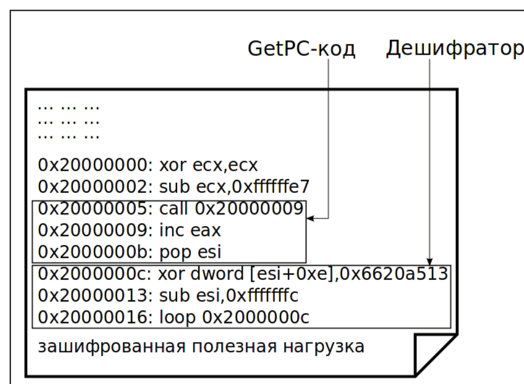


*Рис. 2. Пример структуры зашифрованного шеллкода*

Во-вторых, в исследуемом образце можно искать непосредственно дешифратор [5]. На рис. 2 изображен пример структуры зашифрованного шеллкода, выделены дешифратор и GetPC-код для него. Этот способ тоже имеет вышеупомянутое ограничение, но, в отличие от первого способа, имеет более эффективный, чем перебор, способ решения проблемы нахождения точки входа. В реализациях алгоритма используется второй способ для выполнения этого шага.

### 1.2. Получение расшифрованного образца

Для получения из исследуемого зашифрованного образца его расшифрованной полезной нагрузки также используется эмуляция исполнения инструкций, в ходе которой сохраняются интервалы адресов памяти, к которым были применены операции записи. Эмуляция ведется до конца исследуемого образца или же до встречи ошибки при исполнении очередной инструкции. По окончании эмуляции сохраняется память из адресного пространства эмулятора, взятая по адресам полученных интервалов, — расшифрованный образец.

Далее к полученному расшифрованному образцу снова последовательно применяются первые два шага, поскольку исходный образец мог быть зашифрован несколько раз. Таким образом, к третьему шагу получаем некоторое множество образцов из исходного исследуемого образца.

### 1.3. Удаление мусорных инструкций

Одним из способов обфускации шеллкода является добавление мусорных инструкций, которые не изменяют результата выполнения внедренного кода на целевой машине. К примеру, если в шеллкоде до обфускации не используется какой-либо регистр, то любые операции, затрагивающие только этот регистр, не повлияют на результат исполнения шеллкода, и, таким образом, их можно использовать как мусорные.

На данном шаге алгоритма происходит удаление не только мусорного, но и мертвого (недостижимого) кода. Для упрощения изложения будем называть и то, и другое мусорными инструкциями.

В работе реализовано несколько вариантов удаления мусорных инструкций.

• Для байтовых подходов этот шаг практически опущен. Перед дальнейшим анализом проводится удаление из образцов байтов 0x90 (соответствующих инструкции nop). Несмотря на то что удаление производится побайтово и может быть нарушен смысл многобайтовых инструкций, в состав которых входит байт 0x90, а значит, и смысл всего образца, на корректности алгоритма это не сказывается в силу байтовой ориентированности (в текущей реализации) последнего шага.

• В анализе на основе графа потока управления к построенному графу применяются преобразования по упрощению структуры (удаление подграфов вида jx addr + jnx addr с соответствующими ложными ветвями), а также удаление подряд идущих противоположных инструкций (sub/add, ror/rol, push/pop...) с одинаковыми операндами.

• Анализ на основе построения трасс не использует удаление мусорных инструкций.

В целом, нужно отметить, что в силу специфики дальнейшего анализа удаление мусорных инструкций является необязательным шагом и служит в основном для сокращения длины образца перед последующим более затратным по количеству операций анализом.

### 1.4. Установление подобия

В заключение необходимо определить, подобен ли исследуемый образец какому-либо из известного заданного множества. Т. е. здесь необходимо установить эквивалентность между двумя структурами данных, одна из которых содержит то, что было получено в ходе предыдущего анализа из исходного образца, а вторая  является аналогичным образом полученной структурой для одного из образцов известных шеллкодов. Такую операцию необходимо проделать для всех образцов из множества известных шеллкодов.

В байтовом анализе используются два варианта установления подобия.

В первом из них находится наибольшая общая подпоследовательность. Коэффициент подобия между образцами высчитывается как отношение длины наибольшей общей подпоследовательности к длине известного образца шеллкода. При этом во избежание ложных срабатываний производится разбиение исследуемого образца на блоки, сравнимые по длине с известным образцом шеллкода, с которым производится сравнение.

Еще один байт-ориентированный способ установления подобия заключается в следующем:

1. Один из образцов делится на одинаковые по длине участки (длина в пределах нескольких байтов).

2. Затем производится поиск каждого из участков в другом образце. Подсчитывается количество найденных.

3. Полученное количество делится на общее количество участков в разбиваемом образце — это и есть коэффициент подобия в данном случае.

Такой способ имеет преимущество над поиском наибольшей общей подпоследовательности в случае, если в исследуемом образце была произведена обфускация перестановками блоков кода.

В анализе на основе построения графа потока управления по полученному для исследуемого образца графу формируется структура, представляющая собой список инструкций из базовых блоков при некоторой модификации обхода графа в ширину. Выбор для анализа именно такой структуры данных обосновывается тем, что сравнение самих графов потока управления исследуемого и известного образцов весьма затруднительно. Сравнение графов на изоморфность не приведет к хорошему результату, поскольку структура графа потока управления обфусцированного шеллкода даже после применения деобфусцирующих преобразований может остаться запутанной. Более того, инструкции в базовых блоках в данном случае играют куда более важную роль для сравнения, нежели структура графа. Для сравнения полученных структур снова используется поиск наибольшей общей подпоследовательности, где элементами последовательностей являются

инструкции. При этом инструкции сравниваются без учета используемых в них регистров. Например, инструкции mov eax, 0 и mov ebx, 0 будут считаться равными. При таком способе сравнения будут обнаруживаться шеллкоды, полученные переименованием регистров из исходного. Коэффициент подобия определяется аналогично байтовым подходам.

В ходе анализа на основе трасс по исследуемому образцу строится трасса исполнения (точнее, множество возможных трасс, поскольку в общем случае не известно начало шеллкода в анализируемых данных). Элементами трассы являются ассемблерные инструкции. Длина трассы ограничена константой на количество эмулируемых инструкций. Также при встрече цикла в ходе эмуляции происходит искусственный выход из него после некоторого количества итераций. Полученная трасса сравнивается с аналогично полученными трассами для известных шеллкодов с использованием того же алгоритма нахождения наибольшей общей подпоследовательности по инструкциям с вышеописанной функцией сравнения инструкций. Данный способ имеет преимущество над анализом на основе построения графа потока управления в случае, если произведена обфускация с использованием косвенных переходов (статический анализ в таком случае не применим). Коэффициент подобия определяется аналогично вышеописанным подходам, т. е. полученная длина наибольшей общей подпоследовательности (в инструкциях) делится на длину трассы для известного образца шеллкода.

Надо отметить, что из перечисленных вариантов анализа байтовый является наиболее простым и быстрым по сравнению с другими, поскольку для анализа на основе построения графа потока управления и трасс необходим перебор точек входа. Однако он одновременно является и наименее точным по покрытию возможных полиморфных преобразований. Анализ же на основе трасс является, напротив, наиболее точным, но и наиболее затратным по количеству операций, учитывая затраты на эмуляцию исполнения инструкций.

Данный шаг алгоритма метода наиболее существенен, и его доработка представляет собой основную часть будущей работы.

## 2. Тестирование

В ходе экспериментов с тестовыми данными, сгенерированными в Metasploit Framework версии 4.1.0, реализованный метод показывает точность не меньше 87,33 % (точность определения соответствия необфусцированному образцу). Для обфускации использовались следующие шифраторы: alpha_mixed, alpha_upper, avoid_utf8_tolower, call4_dword_xor, context_cpuid, context_stat, context_time, countdown, fnstenv_mov, jmp_call_additive, nonalpha, nonupper, shikata_ga_nai, single_static_bit. При тестировании на коллекции безвредных документов ошибок второго рода также не выявлено. Таким образом, метод можно использовать для обнаружения в трафике шеллкодов из Metasploit Framework.

*Таблица 1. Результаты тестирования*

| Количество исходных образцов | Количество тестов (все тесты различны и являются полиморфными модификациями исходных) | Правильно распознано |
|---|---|---|
| 14 | 198 | 176 |
| 18 | 221 | 193 |
| 32 | 419 | 368 |

В таблице 1 приведены результаты тестирования для байтового анализа со следующими параметрами:

1.В качестве множества известных шеллкодов взяты полезные нагрузки для ОС Linux, сгенерированные в Metasploit Framework, в качестве тестового множества — их полиморфные

модификации, полученные с помощью доступных шифраторов (шифраторы накладывают ограничения на входные данные, поэтому в тестовых множествах содержится разное количество полиморфных модификаций для разных шеллкодов).

2. В качестве множества известных шеллкодов взяты полезные нагрузки для ОС Windows, сгенерированные в Metasploit Framework, в качестве тестового множества — их полиморфные модификации, полученные с помощью доступных шифраторов.

3. В качестве множества известных шеллкодов взято объединение множеств известных шеллкодов для первого и второго теста, в качестве тестового множества — объединение тестовых множеств шеллкодов для первого и второго тестов.

В силу специфичности тестовых данных наибольшее влияние на ошибки оказали не недоработки в реализации метода, а сбои в работе эмулятора.

Также было проведено тестирование на шеллкодах, представленных в Metasploit Framework в виде исходного кода на языке ассемблер. Их полиморфные модификации были получены с помощью PELock Obfuscator [10]. Тестирование показало наибольшую эффективность анализа на основе трасс. В таблице 2 с результатами тестирования приняты следующие условные обозначения для способов анализа в реализованном детекторе подобия:

- Diff — байтовый анализ на основе нахождения наибольшей общей подпоследовательности;
- N-gram — байтовый анализ на основе поиска блоков из известных шеллкодов (использовались блоки длиной 32 бита);
- CFG — анализ на основе графа потока управления;
- Trace — анализ на основе построения трасс.

*Таблица 2. Результаты тестирования на обфусцированных шеллкодах*

| Реализация | Количество исходных образцов | Количество тестов | Правильно распознано |
|---|---|---|---|
| Diff | 15 | 15 | 5 |
| N-gram | 15 | 15 | 3 |
| CFG | 15 | 15 | 8 |
| Trace | 15 | 15 | 15 |

### 3. Сравнение с другими средствами обнаружения шеллкода

Для сравнения реализованного метода с другими средствами обнаружения шеллкода были выбраны libscizzle [11] и libemu [12], поскольку оба детектора находятся в открытом доступе. libscizzle — библиотека для обнаружения шеллкода, написанная в 2010—2011 г. libemu — библиотека эмуляции исполнения инструкций, которая содержит также свой детектор шеллкодов в числе функций.

Алгоритм работы детектора libscizzle:

- Нахождение GetPC-кода (точнее, последовательности инструкций, предположительно являющейся GetPC-кодом).
- Перебор возможных стартовых позиций шеллкода в окрестности найденного GetPC-кода.
- Эмуляция исполнения инструкций для проверки шеллкода.

Алгоритм работы обнаружения шеллкода в libemu:

- Нахождение GetPC-кода.
- Поиск зависимостей найденного GetPC-кода.
- Эмуляция исполнения инструкций.
- В ходе эмуляции ищется цикл.

Детекторы были протестированы на множествах тестовых файлов, сгенерированных в Metasploit Framework (исходные и зашифрованные образцы), на обфусцированных с помощью PELock Obfuscator образцах, а также на безвредных данных, представленных документами и программами из /usr/bin/.

*Таблица 3. Результаты сравнительного тестирования*

| **Тестовые данные** | **Обнаружено libscizzle** | **Обнаружено libemu** | **Обнаружено детектором подобия** |
|---|---|---|---|
| Исходные образцы | 7/15 | 0/15 | 15/15 |
| Шифрованные образцы | 81/202 | 74/202 | 177/202 |
| Обфусцированные образцы | 13/15 | 1/15 | 15/15 |
| Легитимные документы | 4/235 | 0/235 | 0/235 |
| Легитимные программы | 95/210 | 8/210 | 0/210 |

Однако стоит отметить значительно более высокую скорость работы детекторов libscizzle и libemu, необходимую для сетевого фильтра. Реализованный же в работе метод также может быть использован в качестве сетевого фильтра, но больше подходит для диагностических целей.

**Заключение**

Поскольку атаки внедрения кода составляют существенную часть современных компьютерных атак, методы их обнаружения играют одну из ведущих ролей в информационной безопасности. Постоянно находятся новые уязвимости и, соответственно, появляются новые атаки. В то же время злоумышленники часто используют для атаки известные способы взлома (в данном контексте шеллкоды), видоизменяя их с целью сокрытия. Представленный в работе метод дает возможность обнаружить такие видоизмененные атаки, а также может использоваться для установления соответствия с имеющимися атаками, т. е. в целях диагностики. Эксперименты на тестовых данных, полученных с помощью средства генерации шеллкодов Metasploit Framework, показывают эффективность данного метода. Тестирование, проведенное на обфусцированных шеллкодах, также показывает хорошие результаты реализованного метода при использовании сложных анализаторов.

Все же реализованный метод имеет нацеленность на автоматическую обфускацию, которая используется полиморфными и метаморфными шеллкодами при самораспространении. Вручную можно создать такую эквивалентную версию шеллкода, которая не будет иметь в структуре ничего общего с исходной, а значит, предложенный метод не будет к ней применим. Для обнаружения такого рода полиморфных модификаций потребуется более сложный анализ, включающий в себя методы определения формальной эквивалентности двух программ. Это представляет основное направление будущей работы.

## СПИСОК ЛИТЕРАТУРЫ:

1. *Anley C., Heasman J., Linder F., Richarte G.* The Shellcoder's Handbook: Discovering and Exploiting Security Holes. 2nd Edition. Wiley Publishing INC, 2007.
2. *Булгаков И.А., Гамаюнов Д.Ю., Торощин Э.С.* Обнаружение распространения сетевых червей на основе анализа частоты встречаемости инструкций IA32 в сетевом трафике // Труды третьей всероссийской конференции «Методы и средства обработки информации», МГУ им. М. В. Ломоносова, 6–8 октября 2009, Москва. С. 445–450.
3. *Toth T., Kruegel C.* Accurate buffer overflow detection via abstract payload execution // Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection(RAID), Springer-Verlag Berlin, Heidelberg, Oct. 2002. P. 274–291.

4. *Mason J., Small S., Monrose F., MacManus G.* English Shellcode // Proceedings of the 16th ACM Conference on Computer and Communications Security, Chicago, November 2009. P. 524—533.
5. *Zhang Q., Reeves D. S., Ning P., Lyer S. P.* Analyzing network traffic to detect self-decrypting exploit code // Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS), New York, USA, 2007. P. 4—12.
6. *Polychronakis M., Markatos E. P., Anagnostakis K. G.* Network-level polymorphic shellcode detection using emulation // Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Springer-Verlag Berlin, Heidelberg, July 2006. P. 54—73.
7. *Polychronakis M., Markatos E. P., Anagnostakis K. G.* Emulation-based detection of non-self-contained polymorphic shellcode // Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID), Springer-Verlag Berlin, Heidelberg, September 2007. P. 87—106.
8. *Pasupulati A., Coit J., Levitt K., et al.* Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities // Proceedings of Network Operations and Management Symposium 2004, Washington: IEEE Computer Society, 2004. Vol. 1. P. 235 — 248.
9. *Newsome J., Karp B., Song D.* Polygraph: Automatically Generating Signatures for Polymorphic Worms // Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05), IEEE Computer Society Washington, DC, USA, May 2005. P. 226—241.
10. PELock Obfuscator. URL: http://www.pelock.com/products/obfuscator (дата обращения: 31.03.2013).
11. *Wicherski G.* Linespeed shellcode detection. URL: http://www.honeynet.org/SecurityWorkshops/2011_Paris/Session2_1-Shellcode (дата обращения: 31.03.2013).
12. LibEmu, x86 shellcode detection and emulation. URL: http://libemu.carnivore.it (дата обращения: 31.03.2013).