УДК 004.056.53
DOI: 10.55648/1998-6920-2022-16-3-48-55

Метод защиты программ от отладочных точек останова посредством исполнения фрагментов кода в общем буфере

И. В. Нечта[1]

Работа посвящена проблеме создания антиотладочных механизмов программы. Рассматривается один из наиболее стойких методов постановки точек останова для программ, который не может быть выявлен известными на сегодняшний день алгоритмами. В рамках исследования предлагается новый подход к написанию программ, который приводит к снижению эффективности самого принципа отладки, базирующегося на точках останова. Предлагается хранить функции программы в виде набора байт и перед их исполнением копировать код в один общий буфер. Учитывая, что точки останова привязаны к адресу, мы в результате получим остановку отладчика на каждой выполняемой в буфере функции, а не на какой-то определенной, что существенно увеличит время отладки.

Ключевые слова: антиотладочные механизмы, точки останова, распаковщики, протекторы.

1. Введение

При разработке программного обеспечения автор определяет в лицензионном соглашении условия распространения своего приложения. В одном случае это может быть бесплатная программа с открытым исходным кодом, в другом случае – платная (проприетарная). Часто в лицензионном соглашении указывается запрет не только на любые модификации программы, но и на попытки изучить алгоритмы её работы при помощи отладочных средств (дизассемблеров, декомпиляторов).

Модификация приложений может выполняться, например, с целью нарушения алгоритма проверки лицензионного ключа, когда злоумышленник пытается бесплатно воспользоваться программой. Другим примером может стать попытка встроить стороннюю рекламу в готовый продукт без согласия на то настоящего правообладателя. Если программа широко используема, то модифицированное приложение также будет использоваться и приносить доход постороннему лицу в обход лицензионного соглашения.

Очевидно, что для модификации скомпилированной программы требуется сначала изучить алгоритм её работы. Несмотря на наличие в лицензии явного запрета использовать отладочные средства, разработчики всё же стараются встроить в программу некоторые системы защиты или алгоритмы противодействия известным методам отладки и анализа программ.

Дизассемблер может анализировать программу как в статическом режиме, наглядно представляя код, так и в динамическом режиме, когда она запускается и пошагово выполняется с фиксированным набором входных данных. Очевидно, что статический и динамический анализ взаимно дополняют друг друга.

[1] Исследование выполнено в рамках НИОКТР № 122031600164-6 от 15.03.2022.

Согласно обзору, проведенному в работе [1], существует множество отладчиков, например, `IDA pro debugger`, `Immunity Debugger`, `OllyDebugger`, `Windbg` и другие. Полезным функционалом в современных отладчиках являются:

- поиск сигнатур, позволяющий распознать вызываемые функции программных компонент среды разработки (в статическом режиме);
- интерактивный анализ, использующий значения, передаваемые в регистры и арифметические преобразования в коде программы, для последующего определения особенностей хода выполнения алгоритма (в статическом режиме).
- точки останова (*breakpoints*), используемые при динамическом анализе для остановки программы во время её выполнения.

Развитие отладочных утилит повлекло за собой создание приёмов, препятствующих проведению статического и динамического анализа. Обзор таких подходов представлен в работах [2, 3]. Так, известны программы-*упаковщики* (packers) [4]: `UPX`, `ASPack`, `NsPack` и программы-протекторы: `Themida`, `Armadillo`, `ASProtect`, `ExeCryptor`. Упаковщики ориентированы на то, чтобы сжать и зашифровать программный код в файле, а перед его непосредственным исполнением – распаковать в оперативной памяти и продолжить выполнение. Протекторы в дополнение к шифрованию кода активно противодействуют или завершают работу программы при обнаружении факта работы отладчика.

Наиболее эффективным инструментом в отладке являются точки останова. Аппаратные точки реализованы на уровне процессора и настраиваются через отладочные регистры (*Dr0 – Dr7*). Программные представляют собой процессорную инструкцию с кодом операции `0xCC` (прерывание 3 применяемой в ОС Windows), которая вставляется вместо какой-либо инструкции кода. Когда процессор дойдет до такой перезаписанной команды, то управление перейдет в отладчик. Далее по команде от пользователя восстанавливается перезаписанная инструкция и продолжается выполнение программы. В период такой остановки пользователь может анализировать и изменять фактические значения переменных, регистров, блоков памяти, а также хода выполнения программы.

В качестве меры противодействия перезаписыванию кода протекторы нарушали корректный механизм работы прерываний (через собственную отладку), анализировали фрагменты кода с целью обнаружить вставку чужеродных инструкций и т.д. Тем не менее в работе [5] был предложен подход и программное средство *Spider*, не позволяющее проверить отлаживаемой программе наличие точек останова всеми известными на сегодняшний день способами. Согласно этому подходу производится перехват обращений к страницам памяти (при чтении, записи и исполнении) и подмена страниц. То есть существует две версии страницы в памяти. При обращении к странице в режиме исполнения кода программа предоставляет оригинальную страницу с точкой останова. При обращении к памяти для чтения, например, когда протектор пытается проанализировать код, ему предоставляется ложная страница без точек останова. Таким образом, работая на уровне драйвера операционной системы, предложенная программа реализует «*невидимые точки останова*». Единственным способом обнаружить факт отладки мог бы стать подсчет времени выполнения кода, которое естественно увеличится при отладке. Однако эта возможность нивелируется применением в *Spider* системы виртуализации, реализованной в `IntroLib` [6], когда подменяется время, считываемое командой *RDTSC*.

Упаковщики и протекторы работают, когда к ним на вход подается уже скомпилированный исполняемый файл. У них нет возможности дизассемблировать программу и исправлять вызовы между функциями, кроме случаев обращения к API-функциям через таблицу импорта. Обычно основная работа защитного механизма протектора заканчивается перед передачей управления в оригинальную точку входа программы (*Original Entry Point*). Если злоумышленнику удастся найти *OEP*, то с высокой вероятностью защита будет уничтожена.

В новом подходе защита реализуется на этапе создания исходного кода (до применения протектора) и, соответственно, служит дополнительным уровнем защиты программы.

Так, предлагается новый метод построения программного кода, который существенно осложняет применение точек останова. В дополнение к существующим методам противодействия предлагается выполнять фрагменты кода (функции) в общем буфере. То есть функции хранятся в виде зашифрованных данных и непосредственно перед выполнением они расшифровываются в буфер и там выполняются.

Ранние подходы, реализованные в пакерах [4], уже предполагали расшифровку кода, но позиция расшифрованных функций совпадала с их позицией в оригинальном незащищенном файле[2]. Таким образом, каждая функция была привязана к своему адресу. Использование программных и аппаратных точек останова позволяло отследить факт перехода на нужную функцию при существенной экономии времени (т.е. не задерживаясь в других местах). В настоящем подходе, когда большинство функций выполняется в одном и том же участке памяти, точки останова как программные, так и аппаратные будут возвращать управление в отладчик при выполнении каждой функции (т.е. анализ превратится в трудоёмкую пошаговую отладку). Более того, расшифровка программы происходит мелкими частями, что не даёт возможности одним действием скопировать её образ (дамп) и снять защиту, как это происходит при помощи программ-распаковщиков, например [4, 7].

2. Описание предлагаемого алгоритма

Дадим формальное описание предлагаемого метода. Пусть Prog – множество программ, имеющих одинаковые наборы входных и выходных данных (т.е. различные реализации выполнения некоторой задачи). Обозначим незащищенную программу как $\mathrm{P}\colon \mathrm{P} \in \mathrm{Prog}$. Программы в соответствии с принципом модульного программирования состоят из набора функций (подпрограмм или модулей) $P = \{p_1, p_2, \dots, p_n\}$. Функции попарно не пересекаются $\bigcap_i p_i = \emptyset$. Перед запуском программы они сначала загружаются в память M, и каждая функция получает свой участок памяти $m_i \in \mathrm{M}$, они также не пересекаются между собой $\bigcap_i m_i = \emptyset$. Обозначим процесс загрузки как $\mathrm{Load}\colon \mathrm{P} \to M$.

В рамках исследования требуется разработать метод преобразования программы $Protect\colon \mathrm{P} \to G$, где $\mathrm{G} \in \mathrm{Prog}$, но в отличие от незащищенной программы (с биективным отображением функций в память) в защищенной происходит сюръективное отображение $\mathrm{Load}\colon \mathrm{G} \to M^*$, т.е. $|G| > |M^*|$ и $\bigcap_j m_j^* = \emptyset$. Причем для эффективной защиты требуется выполнение: $|M^*| \to \min$.

Рассмотрим предлагаемый метод по шагам, представленным ниже. В данной статье используются примеры кода на языке программирования C++. Полный исходный код представлен в репозитории [8].

1. Выделяем буфер для записи и исполнения в нем функций, размер которого равен максимальному размеру из этих функций в скомпилированной программе. Блок памяти должен быть выровнен на границу 4 Кб. Например, `buffer=(char*) memalign(4096, CodeSize)`. Размеры функций можно посчитать непосредственно в самой программе (см. ниже) либо их можно посчитать дизассемблером.
2. Устанавливаем атрибуты страницы, где размещен буфер для чтения, записи и исполнения, т.к. область памяти для данных (буфера) не доступна для исполнения кода: `mprotect(buffer, CodeSize,PROT_WRITE|PROT_EXEC)`.
3. Копируем данные в буфер и вызываем функцию.

[2] Речь идет о адресах (смещениях) внутри секции кода, т.к. виртуальный базовый адрес (ImageBase) обычно меняется протектором.

Поясним первый пункт метода. Расчёт размера функции может быть выполнен внутри программы. Например:

```
extern char __start_func[];
extern char __stop_func[];

__attribute__((noinline, section("func")) ) void MyFunction( ){
    // Некоторый код функции
}

int main( ){
    long CodeSize = (long)__stop_func-(long)__start_func;
```

Отметим, что функция находится в секции, название которой выделено жирным шрифтом. `__start_func` и `__stop_func` преобразуются компилятором в константы, соответствующие началу и концу секции.

Для корректной работы алгоритма особое внимание следует уделить компилятору. Так, для Microsoft Visual Studio 2019 вызов функции осуществляется не напрямую, а через дополнительную переходную функцию, см. рис. 1. Здесь мы наблюдаем переходную функцию `sub_4111CC`, которая содержит еще один переход (команду `jmp`) уже на настоящую реализацию интересующей нас функции, реализованной в `sub_412460`. Соответственно, мы не сможем из кода C++ получить ни размер, ни адрес начала искомой функции. Поэтому предлагаемый алгоритм не будет работать с таким компилятором. В настоящем исследовании использовался компилятор *g++ Ubuntu v.9.4.0*.

Выполнение третьего пункта алгоритма – заполнение буфера – не должно вызывать трудностей, т.к. это обычный массив типа char, размещаемый в динамической области памяти. В свою очередь, данные для этого буфера могут быть скачаны по сети, храниться в стороннем файле или в секции исполняемого файла. Данные, очевидно, могут шифроваться. Первичное получение кода функций в виде массива реализовано в примере [8] (файл `Example.cpp`, функция `save()`.

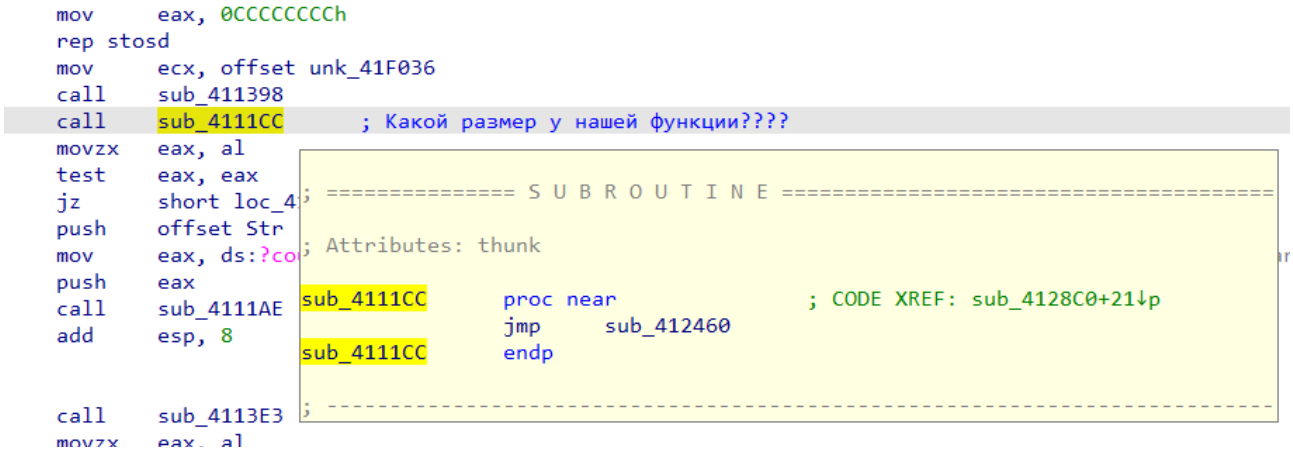

Рис. 1. Применение промежуточной функции перед вызовом основной

Для вызова буферной функции следует создать дополнительные переменные с типом данных – указатель на функцию:

```
typedef void (*PFunc)();
Pfunc* f=(Pfunc*)&buffer;

Load();                                    //загрузка функции в буфер.
f();                                       //вызов функции из буфера.
```

Если внутри функции буфера происходит вызов любой другой дочерней функции (назовем её подпрограммой), то компилятор будет указывать в процессорной инструкции

относительный адрес вида: *переход на несколько байт вперед или назад относительно текущего адреса*. Учитывая, что буфер выделяется в динамической области памяти (т.е. его адрес меняется при каждом запуске программы), следует передавать адрес вызываемой подпрограммы в качестве параметра функции. Например:

```
typedef double (*PFcos)(double x);
typedef double (*Pfunc)(PFcos MyCos);

__attribute__((noinline, section("func")) )
                                        double MyFunction (PFcos MyCos){
    return MyCos(0);
}
void main(){
    Pfunc* f=(Pfunc*)&buffer;
    Load();                                    //загрузка функции в буфер
    double y=(*f)(&cos);      //адрес мат. функции косинус как параметр
}
```

Стоит отметить, что в выбранном компиляторе g++ инициализация локальных переменных осуществляется корректно. Например, для следующего кода содержимое строки представлено в виде констант, что показано на рис. 2. В ряде компиляторов такие строки могут храниться в секции данных, тогда такая инициализация будет невозможна и потребуется передавать строку в качестве параметра.

```
__attribute__((noinline, section("func")) ) int func(PFcos _cos){
char WaterMark[]="Ivan V. Nechta";
    …
}
```

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     [rbp-28h], rdi
mov     rax, fs:28h
mov     [rbp-8], rax
xor     eax, eax
pxor    xmm0, xmm0
movss   dword ptr [rbp-20h], xmm0
mov     rax, ' .V navI' ; Ivan V.
mov     [rbp-17h], rax
mov     dword ptr [rbp-0Fh], 'hceN' ; Nechta
mov     word ptr [rbp-0Bh], 'at'
mov     byte ptr [rbp-9], 0
mov     dword ptr [rbp-1Ch], 0
```

Рис. 2. Инициализация строковых переменных

3. Экспериментальный анализ параметров метода

Для реализации метода внутри защищаемой программы требуется экспериментально подобрать используемые параметры, например, размер буфера. Для проведения эксперимента были отобраны 50 исполняемый файлов (`*.exe`) из каталога имеющихся на компьютере программ `C:\Program Files (x86)`. Учитывая, что программы значительно варьируются по размеру, то все их функции объединялись в один общий набор и анализировались без учета принадлежности к конкретному файлу. Скрипт для анализа функций дизассемблером IDA [9] представлен по ссылке [8] (файл `Analyse.idc`).

В табл. 1 представлены результаты анализа длины функций. Данные сведения нужны для определения длины создаваемого буфера, который должен вмещать в себя любую выполняемую функцию.

Здесь погрешность определялась как $\Delta = \pm\frac{s}{\sqrt{N}} * t_a$, где s – исправленное среднеквадратическое отклонение, N – количество анализируемых функций, t_a – квантиль распределения Стьюдента с уровнем доверия $a = 95\ \%$.

Таблица 1. Результаты анализа длины функций исполняемых файлов

Кол-во функций	Средняя длина, байт	Исправл. СКО	Максимальная длина, байт
83853	180.2 ± 4.2	610	32986

Теперь оценим потенциальную эффективность метода. В формальной постановке задачи было упомянуто, что защищенность достигается при $|M^*| \to \min$. Такое возможно, когда доля функций, размещаемых и исполняемых в буфере, будет значительной. В идеальном случае – все функции выполняются в буфере.

Как уже упоминалось ранее, для вызовов подпрограмм из кода в буфере требуется передача адресов каждой подпрограммы в качестве параметров. Соответственно, простейшей ситуацией является та, при которой вызовов не происходит. Таким образом, был проведен анализ, в котором оценивалась доля функций в исполняемом файле к числу вызываемых ими подпрограмм. На рис. 3 показана доля функций, вызывающих разное количество подпрограмм. Общее количество функций составляет 83853 шт. Например, согласно диаграмме существует всего 1 % функций, у которых внутри не менее 50 вызовов подпрограмм.

Из диаграммы видно, что 32 % функций не делает вообще никаких вызовов (т.е. не требуется передавать дополнительных параметров), а 19 % вызывает только одну подпрограмму (требуется всего один дополнительный параметр). Таким образом, более половины всех функций может исполняться из буфера без существенного усложнения кода программы.

На основании полученных данных мы можем оценить, что в худшем случае для одной требуемой точки останова будут дополнительно возникать $\frac{83853}{50} \approx 1677$ ложных остановок, соответственно, в среднем 838.5 раз, что сделает процесс анализа программы достаточно трудоёмким.

4. Заключение

В настоящей работе предложен метод противодействия отладочным точкам останова. В ходе анализа существующих программ было показано, что метод может быть успешно реализован, т.к. более половины функций либо не вызывают, либо вызывают только одну дочернюю подпрограмму, что делает простой реализацию предлагаемого метода.

Анализ также показал увеличение среднего времени отладки в 839 раз, что является приемлемым результатом. При реализации защиты большое внимание следует уделить используемому компилятору и параметрам его использования. Полученный алгоритм защищает программу не только от статического и динамического анализа, но и от снятия дампа памяти процесса после обнаружения *OEP*.

Рис. 3. Распределение функций по числу вызываемых подпрограмм

Литература

1. *Apostolopoulos T., Katos V., Choo K. K. R., Patsakis C.* Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks // Future Generation Computer Systems. 2021. V. 116. P. 393–405.
2. *Zhang B.* Research Summary of Anti-debugging Technology // Journal of Physics: Conference Series. IOP Publishing. 2021. V. 1744, № 4. P. 042186.
3. *Shields T.* Anti-debugging – a developers view. Veracode Inc., USA, 2010.
4. *Guo F., Ferrie P., Chiueh T. C.* A study of the packer problem and its solutions // Proc. of the International Workshop on Recent Advances in Intrusion Detection, 2008. P. 98–115.
5. *Deng Z., Zhang X., Xu D.* Spider: Stealthy binary program instrumentation and debugging via hardware virtualization // Proc. of the 29th Annual Computer Security Applications Conference, 2013. P. 289–298.
6. *Deng Z., Xu D., Zhang X., Jiang X.* Introlib: Efficient and transparent library call introspection for malware forensics // Digital Investigation. 2012. V. 9. P. S13–S23.
7. Программы-распаковщики [сайт]. URL: `http://www.all-for-rus.narod.ru/unpack.htm` (дата обращения: 21.06.2022).
8. Скрипт для анализа и исходный код примера [репозиторий]. URL: `https://github.com/ivannechta/UntiBPX/` (дата обращения: 21.06.2022).
9. Официальный сайт дизассемблера IDA [сайт]. URL: `https://www.hex-rays.com/ida-free/` (дата обращения: 21.06.2022).

Статья поступила в редакцию 27.06.2022.

Нечта Иван Васильевич
д.т.н., доцент, заведующий кафедрой прикладной математики и кибернетики СибГУТИ (630102, Новосибирск, ул. Кирова, 86), тел. (383) 2-698-216, e-mail: ivannechta@gmail.com.

Method for Programs Protection against Breakpoints by Code Fragments Execution in a Shared Buffer

Ivan V. Nechta
Doctor of technical sciences, Department chairman, Siberian State University of Telecommunications and Information Science (SibSUTIS, Novosibirsk, Russia), ivannechta@gmail.com.

The article is devoted to the problem of creating anti-debugging mechanisms of the program. One of the most robust methods of setting breakpoints for programs is considered which cannot be detected by currently known algorithms. As part of the study, a new approach for program development is proposed which leads to decreasing in the effectiveness of debugging based on breakpoints. It is proposed to store program functions as a set of bytes and copy their code into one shared buffer before executing them. Given that the breakpoints are bound to the address, as a result we will get a debugger stop at each function executed in the buffer, not at any specific one, that will significantly increase the debugging time.

Keywords: anti-debug mechanisms, breakpoints, unpackers, code protectors.

References

1. Apostolopoulos T., Katos V., Choo K. K. R., and Patsakis C. Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. *Future Generation Computer Systems*, 2021, vol. 116, pp. 393–405.
2. Zhang B. Research Summary of Anti-debugging Technology. *Journal of Physics: Conference Series. IOP Publishing*, 2021, vol. 1744. no. 4, p. 042186.
3. Shields T. Anti-debugging – a developers view. *Veracode Inc.*, USA, 2010.
4. Guo F., Ferrie P. and Chiueh T. C. A study of the packer problem and its solutions. *International Workshop on Recent Advances in Intrusion Detection*, 2008, pp. 98–115.
5. Deng Z., Zhang X. and Xu D. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. *Proceedings of the 29th Annual Computer Security Applications Conference,* 2013, pp. 289–298.
6. Deng Z., Xu D., Zhang X. and Jiang X. Introlib: Efficient and transparent library call introspection for malware forensics. *Digital Investigation,* 2012, vol. 9, pp. S13–S23.
7. *Programmy raspakovshhiki* [Programs for unpacking], available at: http://www.all-for-rus.narod.ru/unpack.htm (accessed: 21.06.2022).
8. *Skript dlja analiza i ishodnyj kod primera* [Analysis script and source code example], available at: https://github.com/ivannechta/UntiBPX/ (accessed: 21.06.2022).
9. *Oficial'nyj sajt dizassemblera IDA* [Official site of disassembler IDA], available at: https://www.hex-rays.com/ida-free/ (accessed: 21.06.2022).