Аннотация

Атаки на использующие ошибки переполнения буфера в программах представляют наибольшую опасность для безопасной работы в сети, особенно вместе с приложениями работающими через интернет. Разрушение стека исполнения -- одна из наиболее популярных атак, целью которой является перехват контроля над компьютерной системой. Мы предлагаем метод отслеживания и предотвращения ошибок использующих ошибку переполнения буфера с помощью дополнительного стека возвратных адресов. В отличии от других методов, наш метод не требует модификации существующих программ и дает очень маленькую потерю производительности. Наш метод основан на изменении архитектуры процессора, в нашей статье описывается реализация этого метода с помощью эмулятора процессора SimpleScalar. Результаты показывают, что максимальная потеря при размере дополнительного стека в 64 записи составляет 0.02%, тесты проводятся с помощью SPECINT 2000.

Содержание

1 Введение....4
2 Постановка задачи....8
2.1 Переполнение буфера....8
2.2 Предотвращение атаки использующей ошибку переполнения буфера....9
3 Обзор существующих решений задачи....12
4 Исследование и построение практической части....16
4.1 Реализация разделения стеков основанная на изменении компилятора....19
4.2 Реализации разделения стеков на аппаратном уровне....21
4.2.1 Первое расширение (допущение пополнения стека ложными подсказками) .22
4.2.2 Второе расширение (отсутствие ложных подсказок в стеке)....23
4.2.3 Третье расширение (исключение возможности переполнения стека)....23
5 Описание практической части....25
6 Заключение....30
7 Список цитируемой литературы....31

.

1 Введение

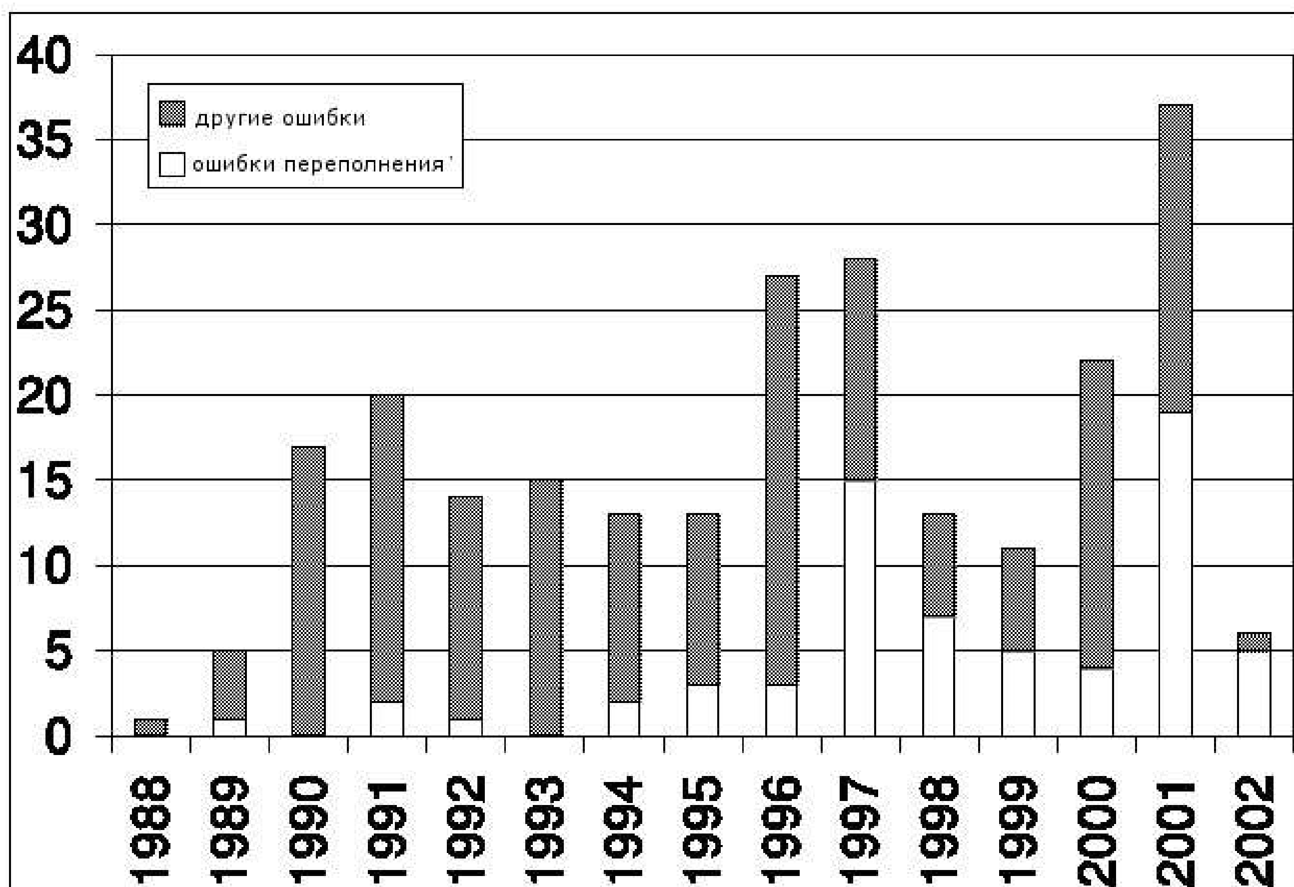


Рисунок 1. Статистика CERT по ошибкам программирования подверженным атакам.

С 1988 года использование ошибки переполнения буфера в сетевом сервисе fingerd вирусом Morris Worm, такие ошибки являлись основой для новых атак. [2, 4, 5]. Puncus и Baker проводят анализ атак основанных на ошибках переполнения буфера. Они приводят несколько примеров таких атак: использование ошибки в реализации SSL [7], Apache Slapper [8], Security Software[9], JPEG Processing (GDI+) [10], и это только некоторые из них.

Для предотвращения таких атак нужно или создавать совершенное программное обеспечение или хранить некоторую мета-информацию, которая находится или в самом ПО или в аппаратной части системы [3]. На данный момент, существует много решений, программных и аппаратных, которые решают эту проблему [11-37, 46-56]. Программное решение, например, часто применяющийся StackGuard [11-14] -- это

хорошее решение для предотвращения атак использующих определенный класс ошибок переполнения буфера. Однако, существующие решения чаще всего не предотвращают основные проблемы ошибок переполнения буфера и только косвенно решают их. Таким образом, они предоставляют лишь частичную защиту чаще всего игнорируя проблемы производительности. Вот некоторые примеры:

- Те методы, которые предотвращают привнесение кода злоумышленника в адресное пространство программы [46, 47], (включая методы реализующие неисполняемые области памяти [15, 16, 17, 18], такие как NX-bit и реорганизация памяти [17, 19, 20]) не могут предотвратить атаки вида return-to-libc, которые не используют привнесенный код и напрямую вызывают функции из библиотек с нужными параметрами [2, 21].
- Существуют методы, которые вставляют специальное значение, в стек прямо перед ячейкой памяти содержащей информацию о возвратном адресе [11, 12, 13, 14]. Этот метод реализует StackGuard. Но после появления StackGuard появились методы обходящие и эту защиту, которые минуют перезаписывание этого специального значения и изменяют возвратный адрес напрямую.
- Методы, которые используют специальный ключ, который генерируется в процессе компиляции или при запуске программы [11, 12, 13, 14, 22, 23, 24], не поддерживают разделяемых библиотек или плохо защищают ключ и алгоритм его генерирующий. Более того, комбинация из атак направленных на утечку информации (так называемые Read Attacks [24]), а так же атаки перебирающие все возможности могут позволить злоумышленнику получить ключ или понять как работает кодирующий алгоритм и, таким образом, получить контроль над системой.
- Те методы, которые контролируют границы доступных областей памяти [33, 34, 35] и реализуются в программном обеспечении имеют существенные недостатки по производительности. Реализация этих методов на уровне аппаратного обеспечения (например, сегментация) [23, 36, 37] приводит к большому количеству маленьких сегментов, что в свою очередь приводит к проблемам с вложенными структурами, и, в некоторых случаях, может накладывать ограничения на модель программирования.
- Те методы, которые используют абстрактные типы данных или предлагают исполнять программы на виртуальных машинах (включая те языки, которые не позволяют программисту интерпретировать значение переменной как тип, к которому

она не принадлежит, так называемые type-safe языки) не защищают программу от атак на переполнение буфера, а просто переносят её на другой уровень. Так как система в любом случае обращается к низкоуровневым библиотекам, операционной системе и аппаратному обеспечению, type-safe языки не защищены от атак переполнения буфера (например, perl [38], java [39, 40], .NET [10]).

- Заключение программ или их частей в специальные ограничивающие окружения (включая, те методы, которые реализованы на аппаратном уровне, например, MS NGSCB [42], TCPA [43, 44], TrustZone [45], Intel La Grande[41]) не предотвращают атак на переполнение буфера. Эти механизмы могут только ограничить ущерб, нанесенный такими атаками.
- Инструменты статического анализа программы [48, 49, 50, 51, 52, 53, 54, 55, 56] могут защитить только от известного набора уязвимостей.

Более того, большинство из этих методов не совместимы с случаями, когда программа выполняется не в non-LIFO режиме (например, используются функции longjmp или происходит обработка сигналов). Мы заключаем, что ни один из предложенных методов не защищает систему от атак на переполнение буфера с целью переписывания возвратного значения в достаточной мере.

Подробный анализ всех типов атак использующих ошибки переполнения буфера показывает, что главной уязвимостью является сохранение целостности адресов (в том числе адресов возврата). Атаки всегда переписывают или содержимое ячейки хранящей указатель на функцию или хранящей адрес возврата с помощью данных полученных из другого окружения (от пользователя, из другого процесса или другого компьютера). Заметим, что защита от вставки кода злоумышленником не предотвращает атаки использующей ошибки переполнения буфера [2]. Таким образом, все можно свести к следующему утверждению: чтобы защитить систему от атак использующих ошибки переполнения буфера, нужно следить за целостностью адресов хранимых в памяти.

Будет показано, что разработанный метод защиты способен предотвратить существующие и будущие атаки на ошибки переполнения буфера с переписыванием адресов возврата. Важной особенностью метода является то, что изменения вносимые в систему не требуют никакого изменения программного обеспечения и 100%-совместимы с существующим ПО. Так же этот метод полностью предотвращает атаки

на переполнения буфера с переписыванием адресов возврата (а не уменьшает вероятность успешной атаки). Явным недостатком является то, что ценой обеспечения этой безопасности является изменение на аппаратном уровне. Однако, эти изменения не велики по сравнению с другими подходами, такими как, например, Intel LaGrande.

2 Постановка задачи.

2.1 Переполнение буфера.

Определение понятия переполнение буфера представлено в определении 1 (из сетевого словаря Webopedia Computer Dictionary [65]):

Определение 1:

Переполнение буфера это состояние, когда данные переданные в буфер превышают вместимость этого буфера и какое-то количество этих данных "залезает" в другой буфер, куда данные не должны были попасть.

Так как буферы могут содержать только какое-то определенное ограниченное данных, когда они полностью заполняются, данные должны попадать куда-то ещё и обычно они попадают в другой буфер, что может привести к разрушению данных, которые уже хранятся в этом буфере.

Воспользовавшись возможностью переполнения буфера злоумышленник может совершить так называемую атаку на ошибку переполнения буфера при определенном стечении обстоятельством. Такая атака может привести к изменению части памяти содержащей другую переменную, информацию о другом процессе (в этом случае произойдет ошибка сегментации) или изменить процесс выполнения программы и выполнить непредусмотренные операции. Основываясь на определении переполнения буфера, определение два определяет понятие атаки использующей ошибку переполнения буфера:

Определение 2:

Атака использующая ошибку переполнения буфера, это атака на компьютерную систему, которая использует операции изменения содержания памяти, чтобы переполнить буфер и изменить данные в ячейках памяти содержащие адреса-указатели на места исполняемого кода. В этих ячейках записываются указатели на программный код помогающий злоумышленнику получить контроль над системой.

Следствие:

Анализ атак использующих ошибку переполнения буфера показывает, что буфер процесса всегда переполняется данными переданными извне (от пользователя, другого процесса, с другого компьютера) -- отсюда и идет опасность.

Чаще всего атакуются ячейки памяти содержащие адреса по которым происходит возвращение, когда функция прекращает свою работу. При изменении адреса и возврате (например, при исполнении инструкции ret на машине архитектуры Intel IA-32) программа продолжит выполнять те данные на которые указывает адрес содержащийся в этой ячейке, однако, указывать она может уже не на тот программный код, который предполагался авторами программы.

2.2 Предотвращение атаки использующей ошибку переполнения буфера.

В этой части мы обсудим необходимые требования для защиты от атак использующих ошибки переполнения буфера.

В атаках использующих ошибки переполнения буфера обычные операции для обращения с памятью используются в уязвимой функции для переписывания адреса (например, адреса возврата).

Из определения 2 следует, что сохранение целостности ячеек памяти содержащих адрес программного кода достаточно для предотвращения атак переполнения буфера. Для большей точности, определим, что такое сохранение целостности ячейки содержащей адрес программного кода в данном контексте:

Определение 3:

Сохранность целостности ячейки памяти содержащей адрес программного кода в памяти означает, что ячейка не изменяется с помощью переполнения буфера данными поступившими извне.

Рассмотрим следствие из определения 3 в свете нашего наблюдения сделанного после введения определения 2, которое отмечало важность того, что атаки приходят

извне: чтобы сохранить целостность ячейки памяти содержащей адрес программного в памяти (например, возвратный адрес), адрес содержащийся в этой ячейке не должен быть порожден из данных, которые пришли извне с помощью переполнения буфера.

Чтобы сохранять целостность, адрес созданный локально может быть помещен в отдельную область памяти, которая не может быть изменена переполнениями буфера (во время инструкций выполнения инструкций call или jump) и перемещен обратно в случае надобности (во время выполнения инструкций return и jump). Работа описывает эти операции, а так же поддержания целостности области памяти, которую не могут изменить переполнения буфера. Введем теорему 1:

Теорема 1:

Изменение содержимого ячейки памяти содержащей адрес программного кода с помощью переполнения буфера данными, которые пришли извне (от другого процесса, пользователя или другого компьютера), является необходимым условием успешной атаки использующей ошибку переполнения буфера.

Доказательство:

Теорема напрямую следует из определений 1 и 2.

Лемма 1.1:

Сохранность целостности ячеек памяти содержащих адрес программного кода в памяти гарантирует защиту от успешных атак использующих ошибку переполнения буфера.

Доказательство:

Из теоремы 1 следует, что для успешной атаки использующей ошибку переполнения буфера злоумышленнику надо изменить содержимое ячейки памяти содержащей адрес программного кода в памяти. Следовательно, предположение

леммы верно.

Итак, возможность успешной атаки, -- это возможность изменить процесс выполнения программы с целью исполнения кода злоумышленника. Чтобы достичь этого, нужно изменить содержимое одной из ячеек содержащей адрес программного кода так, чтобы она указывала на код злоумышленника. Если изменение может быть обнаружено, атака использующая ошибку переполнения буфера может быть обнаружена и остановлена. Так же, если содержимое этой ячейки может быть восстановлено, процесс выполнения может быть продолжен.

Постановка задачи

Разработать алгоритм обнаружения и предотвращения возможности атаки использующую ошибки переполнения буфера, то есть алгоритм, который обнаруживает изменение в определенных местах памяти произведенное атакой и размещает в этих местах памяти изначальные значения.

3 Обзор существующих решений задачи.

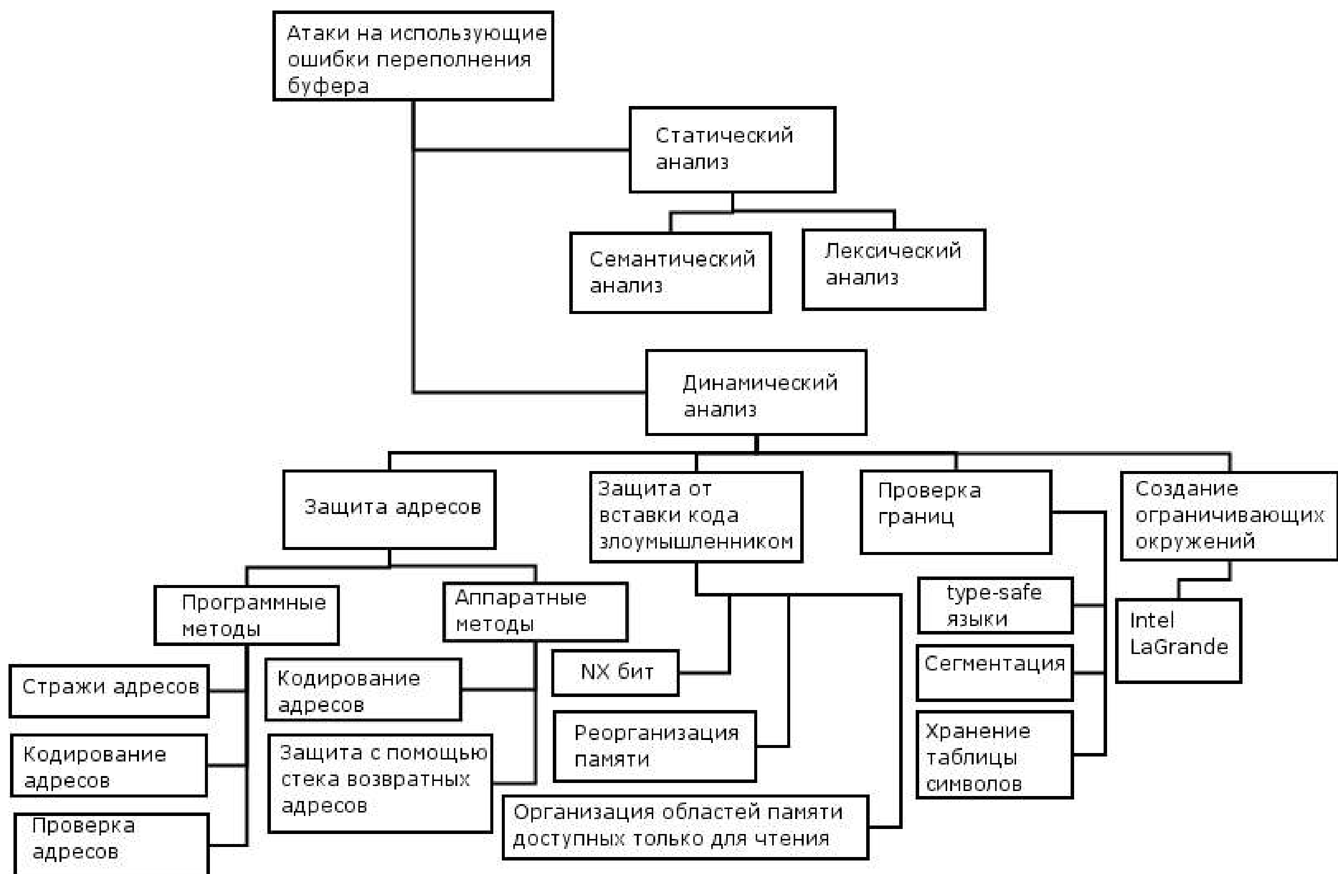


Рисунок 2. Диаграмма существующих методов предотвращения атак использующих ошибки переполнения буфера.

На данном этапе мы ввели определения атаки. Далее будет описаны существующие подходы к защите против атак использующих ошибки переполнения буфера в программах. На рисунке 1 систематизированы различные методы защиты. Для каждого класса из этой диаграммы мы приведем краткое описание механизмов его работы и его потенциальные проблемы. Оценка будет производиться по следующим критериям:

- метод защиты должен в 100% случаев отслеживать атаку изменяющую возвратный адрес функции.
- метод не должен составлять большой дополнительной нагрузки на вычислительную систему.
- метод должен быть совместим с существующим программном кодом и не

требовать его изменения.

Все методы защиты от атак использующих ошибки переполнения буфера в программах делятся на два класса: статический анализ и динамическая защита или предотвращение. К методам статического анализа относятся те методы, которые используют знание о существующих способах атаки и пытаются избавить программу от потенциальных уязвимостей ещё до того, как программа начала использоваться. К динамическим средствам обнаружения атаки и защиты от неё относятся те методы, которые используют лишь знание о среде выполнения.

Чаще всего, средства статического анализа [49, 50, 51, 52, 53, 54, 55, 56] исследуют исходный код и пытаются обнаружить возможные уязвимые места, в которых может произойти переполнение буфера. В основном, эти средства обрабатывают и анализируют исходный код до компиляции учитывая специальные указания насчет исходного кода, данные им пользователем. Результатом работы является отчет с предложениями по модификации программы. В данном случае может использоваться как простой механизм распознавания образов [49], так и лексический [50, 51] и синтаксический [52] анализаторы. Средства основанные на синтаксическом анализе обычно лучше различают правильное и неправильное использование функций работающих с памятью, чем те, что основаны на лексическом анализе или распознавании образов. Преимущество статического анализа заключается в том, что он позволяет программисту избавиться от многих ошибок перед тем как использовать программу. Однако, средства статического анализа не имеют никакой информации о процессе выполнения и поэтому не смогут учесть все возможные проблемы. Вне зависимости от того, насколько хорошим является средство статического анализа, программист принимает решение о том, как исправлять логику работы программы.

Средства динамической защиты от атак могут быть далее разделены на четыре класса: средства защиты адресов, защиты от вставки кода злоумышленником, проверки грани и создания ограничивающих окружений.

Средства защиты адресов или отслеживают изменение адресов или просто не дают возможности изменить некоторые из них. Это можно сделать с помощью помещения специального значения-стража в ячейку идущую перед ячейкой с адресом [11, 12, 13, 14, 20], с помощью кодирования адреса [22, 23, 24] или с помощью сравнивания этого адреса с его резервной копией [25, 29, 30, 31, 32]. После помещения

специальное значение-стража в ячейку идущую перед ячейкой с адресом злоумышленник может напрямую исправить адрес без изменения стража, как это показано в статье Bulba и Kil3r [59]; так же можно воспроизвести это значение в самом буфере [60]. При использовании метода кодирования адреса возникают проблемы с хранением кодирующего ключа и алгоритма [59, 60]. В любом случае, ключ или значение-страж созданные в процессе компиляции программы не изменяются при запуске, а значит их значение можно легко получить с помощью де компиляции программы. В случае, когда ключ или значение-страж генерируются при каждом запуске, проблемы с хранением ключей все равно остаются, примеры атак прорывающих подобную защиту описаны Newsham [60] и Litchfield [61]. В случае, когда адрес сравнивается с его резервной копии при его использовании [25, 26, 27, 29, 30, 31, 32], уязвимым местом программы остается безопасное хранение резервных адресов, как это показано в [59].

Средства защиты от вставки кода злоумышленником или предотвращают саму вставку или предотвращают выполнение этого кода. Механизмы, которые позволяют это сделать, часто реализуются на уровне аппаратного обеспечения, например NX-bit позволяющий сделать некоторые области памяти не исполняемыми [15], или на уровне операционной системы (например, в Solaris, BSD и Linux [17, 18]). Так же можно воспользоваться техниками реорганизации памяти [17, 19] и кодирования инструкций [46, 47]. В любом случае, эти средства не позволяют защитить программу от return-to-libc атак, которые используют уже загруженные в память функции для получения контроля над системой.

Разработаны специальные механизмы не дающие буферу переполниться, которые следят за его границами при каждом доступе к памяти. Один из вариантов реализации такого метода заключается в следующем: для каждого указателя на область в памяти определяется базовый указатель на начало буфера, тогда указатель может принимать значения из некоторой области, которая зависит от этого базового указателя. Каждый раз при попытке доступа к памяти производится проверка, куда ссылается указатель, по которому идет обращение, в случае, если указатель выходит за границы определенные своим базовым указателем возникает ошибка. Этот метод в принципе защищает от атак с переполнением буфера. Однако, потеря производительности программы использующей этот метод может составлять 2900% в

том случае, когда исходный код написан с активным использованием указателей [33]. В результате, подобные средства идеальны для отладки, однако они не могут быть использованы во время её использования. Так же, можно использовать библиотеку позволяющую работать со строками и не допускающую переполнения их буферов [29], однако это требует модификации уже существующего исходного кода. В случае использования type-safe языков программирования (например, Java, PERL или .Net) процесс проверки на выход за границы областей памяти происходит в виртуальной машине. Чаще всего, виртуальная машина пишется на C/C++ так как она должна взаимодействовать со стандартными библиотеками. Программы написанные на type-safe языках все же могут быть атакованы, хотя вероятность такой атаки снижается. До сих про происходят успешные атаки использующие ошибку переполнения буфера в Java [39, 40], Perl [38], .Net [10].

Создание ограничивающих окружений, это механизм основанный на определенной политике безопасности. Так как переполнение буфера происходит, когда информация из одной функции передается в другую, окружение не может защитить от атак использующих переполнение буфера. Введя правильную политику безопасности возможно ограничить возможный урон причиненный атакой. Ограничивающие окружения могут быть реализованы на нескольких уровнях: на уровне ядра [63], на уровне пользователя [62, 63, 64] или на даже на уровне аппаратного обеспечения [41, 42, 43, 44, 45].

Можно заключить, что ни один из приведенных механизмов не позволяет эффективно и полностью защитить программу от атак использующих ошибку переполнения буфера, в том числе, атак модифицирующих возвратный адрес.

4 Исследование и построение практической части.

```
char shellcode[]  = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
                    "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
                    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
                    "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
                    "\x80\xe8\xdc\xff\xff\xff/bin/sh";
char large_string[128];
void main() {
  char buffer[96];
  int i;
  long *long_ptr;

  long_ptr = (long *) large_string;
  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;
  for (i = 0; i < strlen (shellcode); i++)
    large_string[i] = shellcode[i];
  strcpy (buffer, large_string);
}
```

В начале этой части будут приведены примеры атак, которые могут позволить злоумышленнику получить контроль программными системами. На листинге приведен пример программы реализующей многоуровневую атаку использующую ошибку переполнения буфера. Сначала атакующая программа подготавливает данные для ввода в переменной large_string. Затем, она копирует их содержимое в buffer в стеке. Заметим, что переменная buffer занимает в памяти 96 байт, а переменная large_string -- 128 байт. Функция strcpy полностью копирует содержимое large_string в переменную buffer без проверки нарушения грани, таким образом происходит переполнение буфера. Содержимое large_string специально определено так, что одновременно происходит и вставка вредоносного кода и изменение возвратного адреса в стеке.

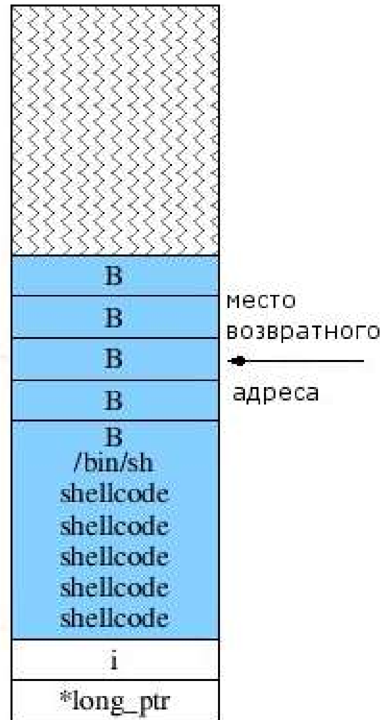


Рисунок 2. Переполнение буфера в стеке.

Содержимое стека выполнения до и после исполнения функции strcpy, копирующей значение large_string, показано на рисунке 2. Область памяти закрашенная голубым цветом на рисунке изменяется при вызове функции strcpy и переполнении буфера buffer. Первая часть буфера заполнена содержимым из переменной shellcode, которая и содержит вредоносный код. Вторая часть часть заполнена адресом этого вредоносного кода, то есть адресом переменной buffer. Когда функция main возвращает значение контроль переходит по адресу переменной buffer и начинается исполнение вредоносного кода. Вредоносный код исполняет системный вызов execv, чтобы запустить командную оболочку.

Приведенный пример является искусственной моделью. На самом деле для успешной атаки вредоносный код помещают в переменную окружения, его вводит пользователь или же он получается через сетевое соединение. Успешная атака на процесс, например на setuid программу или системный сервис уже запущенный с правами администратора, даст возможность атакующему получить контроль над командной оболочкой с правами администратора. Такие атаки иногда основываются на

восстановлении структурной схемы и алгоритма программы по машинным кодам атакуемой программы и такую атаку сложно осуществить. Однако, используя методы предлагаемые в статье [1] спроектировать её становится гораздо проще.

Разделение контрольного стека и стека данных.

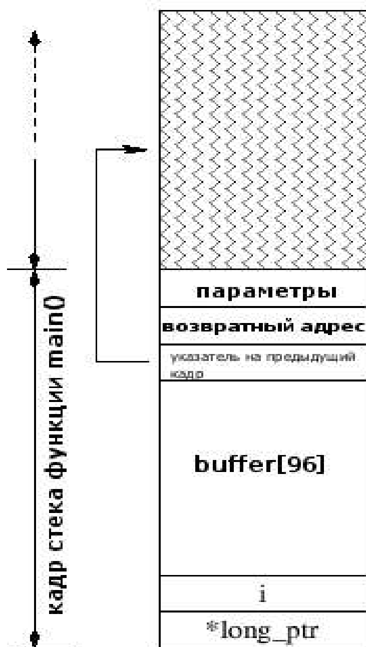


Рисунок 3. Единый стек данных.

Основной причиной, по которой возможны атаки на существующие системы, это общий стек хранения локальных данных функций (временные буферы, переменные и параметры вызова) и данные управляющей логики программы (возвратные адреса), это показано на рисунке 3. Когда функция получает данные и передает их в буфер выделенный в стеке без проверки грани и сам буфер и возвратный адрес (который находится выше по стеку) могут быть переписаны.

Предлагаемый подход заключается в разделении стека на два: стек содержащий данные управляющей логики программы, который содержит исключительно возвратные адреса, и стек локальных данных, который содержит временные данные и аргументы функций (как показано на рисунке 3).

Сначала мы опишем реализацию этого подхода на уровне программного обеспечения с помощью изменения компилятор, а затем опишем возможные

изменения в архитектуре процессора для реализации подхода на аппаратном уровне.

Кратко опишем способы вызова функций на архитектуре IA-32, так как реализация подхода происходила именно на этой платформе. В архитектуре IA-32 операторы call и return реализованы с помощью двух инструкций, call и ret соответственно. Во время исполнения инструкции call процессор помещает возвратный адрес (чаще всего, это адрес следующей после call инструкции) в стек расположение в памяти которого определяется стековым регистром esp. При вызове инструкции ret процессора достает возвратный адрес из стека, на который указывает esp и делает по нему безусловный переход. Когда вызывается инструкция ret процессор предполагает, что в вершине стека находится правильный возвратный адрес. Компилятор использует этот же стек для помещения туда параметров функций и заведения места для локальных буферов и переменных.

4.1 Реализация разделения стеков основанная на изменении компилятора

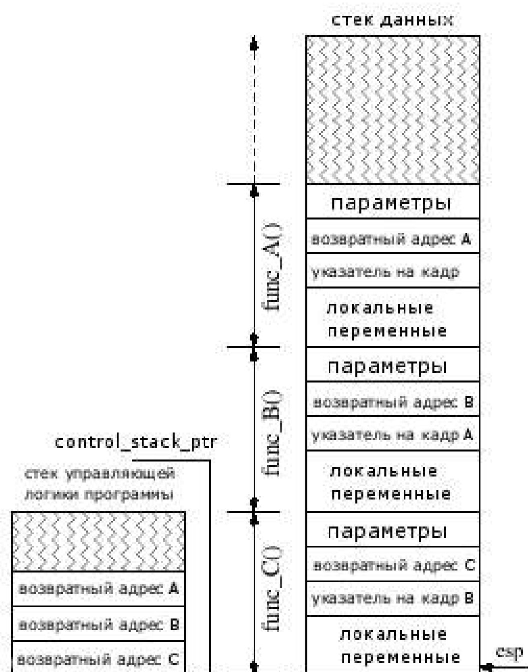


Рисунок 4. Реализация дополнительного стека на программном уровне.

Реализация подхода основанная на изменении компилятора проиллюстрирована на рисунке 4. В начале исполнения каждый функции, возвратный адрес, который

сохраняется в стеке локальных данных, так же сохраняется в стеке данных управляющей логики программы. Перед тем как возвратиться их функции значение из стека данных управляющей логики записывается в стек локальных данных. Таким образом, инструкция ret берет из стека то значение, которое находится в стеке данных управляющей логики, а не то, что было в стеке данных по завершению выполнения функции. Как показано на рисунке 4, во время работы функции в памяти находятся две копии возвратного адреса, одна находится в стеке данных и может быть переписана переполнением буфера и другая, которая находится в стеке данных управляющей логики. Так как инструкция ret всегда использует ту копию, которая не может быть переписана, переполнение буфера не может нарушить процесс исполнения программы.

Проверочная программа	***Максимальная вложенность вызовов функций***
bzip2	11
crafty	28
eon	30
gap	362
gcc	32
gzip	14
mcf	41
parser	62
perlbmk	17
vortex	29
twolf	15

Таблица 1.Максимальные вложенности вызовов функций для проверочных программ входящих в SPECint 2000

Преимущества такого подхода состоят в том, что он не требует дополнительных изменений процессора. Недостатки заключаются в потере производительности. Так же, существуют некоторые проблемы, которые не были учтены при создании этой реализации:

- Размер стека данных управляющей логики. Размер области памяти занимаемой стеком данных управляющей логики зависит от количества вложенных вызов функций, которые выполняются программой. На таблице 1 приведены максимальное

количество вложенных вызовов для программ входящих в пакет SPECINT 2000. Большинство из этих приложений делают мало вложенных вызовов (меньше 32). Так, стека занимающего одну страницу памяти будет достаточно почти для всех приложений использующихся на практике.

- Поддержка приложений с несколькими нитями исполнения и поддержка longjmp. Для приложений с несколькими нитями исполнения, каждая нить должна иметь свой стек данных управляющей логики. Для решения этой задачи можно изменить функцию pthread_create, которая создает нити исполнения, так, чтобы она выделяла нужное место для стека данных программной логики. Setjmp и longjmp функции используются программами для возврата сразу из нескольких уровней вложенных вызовов. Если добавить поле к структуре, в которой хранится состояние программы на момент вызова функции setjmp, в котором будет сохраняться а потом восстанавливаться адрес стека данных управляющей логики, проблема будет решена.

Реализация подхода основанная на изменении компилятора похожа на реализацию StackShield[25]. Существенное различие заключается в том, что StackShield меняет ассемблерный код после того, как он будет регенерирована компилятором. Реализация описанная ниже изменяет компилятор, поэтому оптимизатор компилятора может оптимизировать и код сохранения и восстановления данных из стека данных управляющей логики программы.

4.2 Реализации разделения стеков на аппаратном уровне

Реализации подхода на аппаратном уровне так же основывается на создании отдельного стека данных управляющей логики, однако в отличии от реализации с изменением компилятора, не переписывает возвратный адрес, который хранится в стеке локальных данных, а обнаруживает атаку после того как адрес в стеке был изменен, но до того, как это могло привести к каким-либо негативным последствиям.

Работа такого нового стека данных управляющей логики похожа на подход используемый в стеке возвратных адресов описанный в статье [17], который реализован в большинстве современных процессоров. Стек возвратных адресов используется процессором на стадии выборки инструкций, чтобы оптимизироваться

эффективность использования канала между памятью и процессором. Стек возвратных адресов может предсказать адрес по которому может последовать возврат. Когда процессор загружает инструкцию вызова функции возвратный адрес помещается в стек возвратных адресов. Когда процессор загружает инструкцию возврата из функции возвратный адрес выгружается из стека. Затем по этому адресу определяется откуда должна быть загружена следующая инструкция. Стек возвратных адресов используется как подсказка процессору, информация полученная из него оказывается верна более чем в 99% случаев [17]. Можно заметить, что стек возвратных адресов содержит дополнительную копию возвратных адресов в стеке процессора и там не может произойти переполнения. Лишняя копия возвратного адреса может быть использована для отслеживания ситуаций, когда возвратный адрес в стеке локальных данных был изменен. Однако, стек возвратных значений может выдавать ошибочные подсказки из-за рискованного обновления стека и перегрузки из-за ограниченного размера. Атака использующая ошибку переполнения буфера тоже может повлиять на ошибочной подсказок содержащихся в стеке, так как может изменить точный адрес возврата в стеке.

Предлагаются три возможных расширения механизма действия стека возвратных адресов. Первое расширение заключается в том, что процессор будет выдавать ошибку всякий раз, когда из стека выдается неверная подсказка. Второе расширение предлагает дополнительный стек, который используется процессором на стадии фиксации (а не на стадии выборки, как обычный стек). Это расширение устраняет вероятность неверной подсказки. В третьем расширении, предлагается новый стек, который так же используется процессором на стадии фиксации и механизмы работы с ним предотвращают его переполнение.

4.2.1 Первое расширение (допущение пополнения стека ложными подсказками)

Механизм обнаружения ложных подсказок полученных из стека возвратных адресов может быть расширен для обнаружения атаки использующей ошибку переполнения буфера. В случае первого предложенного расширения, всякий раз, когда в стеке находится неверная подсказка, процессор выдает ошибку и операционная система может её обработать и определить, произошла ли атака или стек выдал неверную

подсказку под другой причине. Обработчик операционной системы может воспользоваться таблицей возможных точек в памяти, куда может произойти возврат, или проанализировать последовательность вызванных функций внутри программы. В случае, если произошло переполнение стека, обработчик не сможет сразу отследить предыдущее нормальное состояние стека, чтобы продолжить нормальное выполнение программы. Для того, чтобы найти предыдущее состояние обработчику придется провести анализ всего стека проходя по возвратным ссылкам. На тестовой реализации в эмуляторе процессора эта операция может занять 450 циклов процессора (Пусть, глубина вложенности вызовов равна 25, это пессимистический прогноз, судя по таблице 2. Отслеживание каждого уровня требует прочтение сохраненного указателя на кусок стека, относящейся к этому уровню. В нашем эмуляторе, одно прочтение занимает 18 циклов, а значит потеря производительности составит примерно 450 циклов процессора). Когда размер стека составлял 64 записи, большинство приложений из набора SPECINT 2000 выполнялись с потерей производительности в 100%. Главной причиной этой потери оказалось то, что процессор пополнял стек значениями на стадии выборки. Чем чаще из стека доставались правильные подсказки, тем ниже была потеря производительности.

4.2.2 Второе расширение (отсутствие ложных подсказок в стеке)

В случае второго предложенного расширения, вводится дополнительный стек. Так как пополнение стека процессором происходит на стадии фиксации, туда не попадают ложные подсказки. При такой реализации, обнаружение ложных подсказок (которые появились не из-за переполнения буфера и переписывания части стека локальных данных) может произойти лишь при переполнении стека возвратных значений, в этих случаях вызывается обработчик. Учитывая информацию из таблицы 1, можно сказать, что при размере стека в 64 потеря производительности будет происходить лишь на программе gap, так как вложенность вызовов в ней сильно превышает 64.

4.2.3 Третье расширение (исключение возможности переполнения стека)

Чтобы улучшить производительность системы, мы будем основываться на стеке из второго предложенного расширения. Как было показано, потеря производительности происходит при переполнении стека возвратных адресов. Правильная реализация нашего подхода должна создавать потерю производительности лишь в том случае, когда происходит переполнение стека. Для достижения этого результата достаточно устранить две причины ложных подсказок, которые выдаются даже в случае отсутствия переполнения буфера:

- преждевременное пополнение (подсказка оказывается ложной из-за ложного предсказания условного перехода)
- переполнение стека (в случае, когда размер стека меньше чем глубина вложенности вызовов функций. Первая причина была устранена в предыдущей секции. Чтобы устранить возможность переполнения стека, можно сохранять часть стека в соответствующую область в памяти выделенную в адресном пространстве каждого процесса. Освобожденное место в стеке можно использовать для возвратных адресов порожденных новыми вызовами функций. Очевидно, что после нескольких возвратов из функций сохраненную часть стека в памяти надо восстанавливать. Сохранение и восстановление частей стека может быть реализовано как на уровне операционной системы, так и с помощью специального устройства в процессоре.

5 Описание практической части.

Для того, чтобы оценить реализации разделения стеков на аппаратном и программном уровне и показать совместимость с существующим исходным кодом поддержка дополнительного стека была реализована в наборе компиляторов GNU (GNU Compiler Collection) и эмуляторе процессора SimpleScalar[66] соответственно.

В случае реализации дополнительного на программном уровне, компилятор дает указания загрузчику программы выделять место для стека данных управляющей логики и хранит адрес этого стека в специальной переменной control_*stack_ptr.* Сохранение возвратного адреса реализовано в пред функции, которая выполняется в начале выполнения каждой функции. Эта пред функция так же отвечает за организацию стека, инициализацию стекового регистра, сохранением текущих значений регистров и выделением памяти для временных переменных. Эта пред функция генерируется для каждой функции в процессе компиляции. Изменение состоит в добавлении программного кода копирующего возвратный адрес из стека локальных данных в стек, на который указывает переменная control_stack_ptr.

Эффективность этого подхода проверялась с помощью компиляции программ, которые имитируют атаку использующую ошибки переполнения буфера, входящих в пакет LibSafe [29]. В пакет входит 5 программ, которые вводят специально подготовленную строку на вход и одну настоящую атаку, атаку на программу xlockmore, которая закрывает доступ к терминалу компьютера в системе X Window. Атака на xlockmore используют известную уязвимость в этой программе, а так как программа исполняется с правами администратора, злоумышленник может получить доступ администраторского уровня. Если скомпилировать эти программы с не измененным компилятором, в результате выполнения выходных файлов будет запущена программная оболочка. При компиляции программ исправленным компилятором программы завершили свое выполнение с ошибкой не дав потенциальному злоумышленнику получить доступ к системе.

Проверочная программа	***Обычное время***	***Дополнительный стек***	***Потеря производительности***
gzip	356.6	383.5	8.12%
vpr	593.2	616.1	3.65%
gcc	337.9	347.3	3.52%
mcf	707.6	707.7	0.01%
crafty	407.9	466.1	14.37%
parser	486.6	515.7	5.70%
gap	240.8	277.8	15.21%
vortex	505.3	626.4	23.79%
bzip2	496.4	532.7	7.33%
twolf	1125.5	1155.7	2.58%
ftp (1kb)	0.115	0.125	5.44%
ftp (1mb)	0.125	0.131	2.35%

Таблица 2. Потеря производительности при реализации дополнительного стека возвратных адресов на программном уровне.

Потеря производительности при применении программной реализации разделения стеков измерялась с помощью программ тестирования SPECINT 2000 и стандартного набора вводных данных. Это набор программ измеряющих производительность архитектуры процессора и памяти. Так же для тестирования использовался ftp-сервер wu-ftpd. В таблице 2, в колонке "обычное время" показано время затраченное на исполнение программ скомпилированных с не измененным gcc. В колонке "дополнительный стек" показано время затраченное на исполнение проверочных программ скомпилированных измененным компилятором. В таблице показано, что потеря производительности при запуске программ проверяющих производительность вычислений с целыми числами колеблется между 0.01% и 23%. Информация о производительности ftp-сервера получена с помощью измерения промежутка времени между началом и концом загрузки файла на ftp-сервер. В таблице приведено два теста: в первом случае клиент загружает файл размером 1 kb, а во втором 1 mb. Потеря производительности в случае загрузки 1 kb на сервер составляет около 5%, а для 1mb -- 2%. В случае большего размера файла больше времени тратится на операции ввода вывода, что и служит причиной уменьшения потери производительности из-за введения дополнительного стека.

Полученный проигрыш в производительности получен из-за более частого доступа к памяти: при вызове функции происходит происходит две операции чтения и две операции записи в память (чтение из стека данных, запись в стек управляющей логики программы, чтение control_stack_ptr и запись нового значения). Похожая ситуация происходит и при возврате из функции и там требуется ещё четыре операции с памятью. Итак, для выполнения каждой функции требуется 8 лишних доступов к памяти, в таком случае может быть заметная потеря производительности. Однако, для северных приложений эта проблема не столь актуальна, так как потеря производительности не будет столь велика.

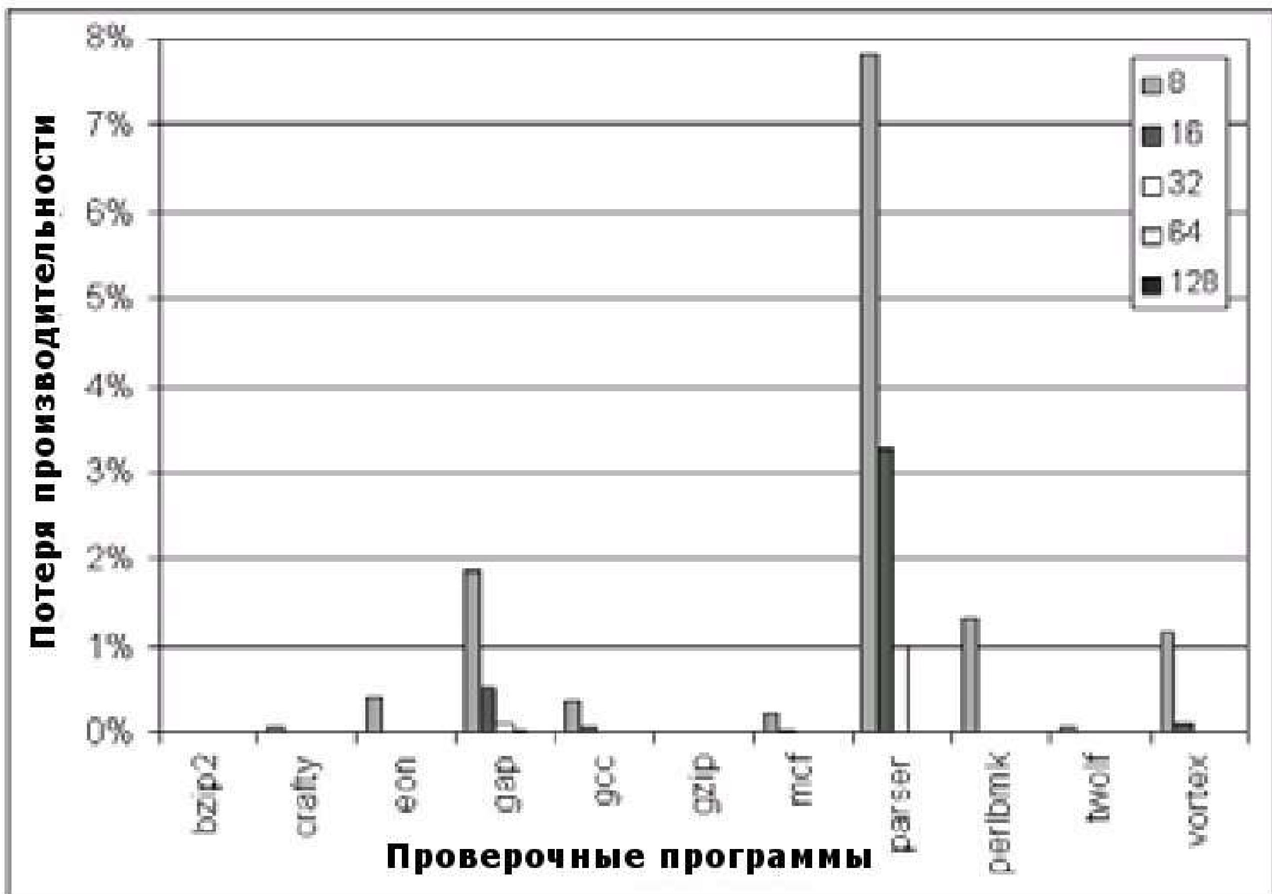
Пробная реализация дополнительного стека на аппаратном уровне была выполнена на основе имитатора процессора SimpleScalar. SimpleScalar, это имитатор используемый для моделирования нововведений в архитектуре процессора и является в этой области стандартом. В нем есть специальные средства для отслеживания работы составных частей процессора, таких как иерархии внутрипроцессорной памяти (кэша) и внеочередного исполнения.

Для SimpleScalar был разработан дополнительный стек, размер которого передается в качестве параметра. С помощью программ поставляемых с дистрибутивом SimpleScalar измерялись показатели производительности процессора.

Основной причиной потери производительности измененного процессора с оказалась проблема переполнения стека управляющей логики программы. В созданной реализации каждый раз, когда стек переполняется половина его содержимого перемещается в структуру находящуюся в памяти. Пусть, *n* – число циклов процессора затрачиваемых на перемещение одной записи из стека в память, а s размер стека, а p, это лишняя затрата циклов процессора связанная с переполнением стека. Тогда,

$$p = n * (\frac{s}{2})$$

Во время проверки число n оказывается равно задержке обращения к памяти (по умолчанию, в SimpleScalar это число равно 18). Эта формула является пессимистическим прогнозом, так как только часть перемещений записей будут выполнятся с задержкой, большинство из них будут перемещены в одном пакете. С другой стороны, формула не учитывает то, что при переполнении стека циклы процессора могут быть затрачены на обнаружение структуры данных где хранятся



выгруженные части стека в памяти. Учитывая эти факты можно сказать, что формула является приемлемой для оценки общей потери производительности в случае переполнения.

Рисунок 5. Потеря производительности в зависимости от размера стека.

Потеря производительности при использовании дополнительного стека оценивалась с помощью тестов SPECINT 2000. На рисунке 5 показана потеря производительности в зависимости от размера дополнительного стека. Большинство проверяемых программ исполняются с небольшой потерей производительности, когда размер стека управляющей логики программы гораздо меньше их максимального уровня вложенности вызова функций. Только программы gap и parser исполняются с заметной потерей, когда размер дополнительного стека равен 32 записям, а максимальная вложенность вызова функций равно 362 и 62 соответственно. При размере стека равном 64 записям только gap исполняется с небольшой потерей производительности (0,02%, 0,005% при размере 64, 128 и 256 соответственно).

Потеря производительности пропадает при размере стека равно 512 записям. Из результатов можно сделать вывод, что размера в 64 записи будет достаточно для того, чтобы потеря производительности при исполнении большинства приложений была минимальной.

При реализации стека управляющей логики программы внутри имитатора процессора SimpleScalar не были рассмотрены некоторые особенности реального использования процессора. Так, например, при каждом переключении контекста (при переключении между процессами), ядру операционной системы придется сохранять содержимое всего стека в структуре соответствующей одному из процессов и восстанавливать содержимое стека из структуры соответствующей другому. Потеря производительности связанная с переключением контекстов не была измерена настоящей реализацией. Однако, скорее всего, при реализации поддержки переключения контекстов не надо будет сохранять и восстанавливать весь стек так как программы редко достигают своего максимального уровня вложенности вызова функций. Другие проблемы могут быть связаны с поддержкой функций setjmp и longjmp (они были описаны выше), они могут быть решены с помощью добавления отдельной инструкции, которая восстанавливает стек до определенного уровня.

6 Заключение.

В дипломной работе описан алгоритм обнаружения и предотвращения атак использующих ошибки переполнения буфера. В этой подходе выделяется отдельный стек для хранения возвратных адресов, называемый стеком управляющей логики программы. В случае программной реализации, в стеке, находящимся в оперативной памяти системы, хранится дополнительная копия возвратного адреса, которая используется при возврате из функции. Таким образом, даже если злоумышленник переписывает возвратный адрес в стеке данных программы он не может получить доступ к системе исполняющей эту программу. Программная реализация была сделана на основе набора компиляторов GNU. При оценке производительности потеря оказалась достаточно небольшой (2% для сервера ftp). При аппаратной реализации процессор хранит часть стека в процессорной и обнаруживает атаку при исполнении инструкции возврата из функции. Максимальная потеря производительности при запуске программ на имитаторе такого процессора составляет 0.02% при размере стека равном 64 записям. При проверке программ использовались тесты SPECINT 2000 без изменений, тем самым было продемонстрировано, что обе реализации полностью совместимы с существующими программами и не требуют их изменения.

7 Список цитируемой литературы

1 C. Schmidt, T.Darby. The What, Why, and How of the 1988 Internet Worm, [HTML] http://www.snowplow.org/tom/worm/worm.html

2 J.Pincus, B. Baker, Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns // IEEE Security & Privacy, Vol. 2. No. 4. July/August 2004. P. 20 -27

3 Andy Glew, "Segments, Capabilities, and Buffer Overrun Attacks," Computer Architecture NEWS // ACM SIG Computer Architecture Vol.31. No. 4. September 2003. P. 26-31

4 E. Chien and P. Szr (Symantec Security Response), Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses // Virus Bulletin Conference, (Sept. 2002)

5 Aleph One, Smashing stack for fun and benefit, // Phrack Magazine, 49(7), Nov.1996

6 J. Viega, G. McGraw, Buffer Overflows, Chapter 7, Building Secure Software, Addison Wesley, pp.135-185 (2002)

7 OpenSSL Security Advisory [30 July 2002], [HTML] http://www.openssl.org/news/secadv_20020730.txt

8 S. HsiangRen, Apache/mod_ssl (slapper) Worm, GIAC Certified Incident Handler (GCIH), [HTML] http://www.giac.org/practical/HsiangRen_Shih_GCIH.doc (Oct. 2002)

9 D. Geer, Just How Secure Are Security Products?, IEEE Computer Vol.37, No. 6 - June 2004, pp. 14-16

10 Microsoft Security Bulletin MS04-028: Buffer Overrun in JPEG Processing (GDI+), [HTML] http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx (Oct, 2004)

11 C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, StackGuard: Automatic Adaptive Detection and Prevention of Buffer- Overflow Attacks // the proceedings of the 7th USENIX Security Symposium (Jan. 1998)

12 C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard // proceedings for Linux Expo, Raleigh, NC (May 1999)

13 C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. // proceedings for DARPA Information Survivability Conference and Expo (DISCEX), Hilton Head Island SC, (Jan. 2000)

14 H. Hinton, C0 Cowan, L. Delcambre, and S. Bowers, SAM: Security Adaptation Manager," // the Annual Security Applications Conference (ACSAC), Phoenix, AZ, (Dec. 1999)

15 T. Krazit, PCWorld - News - AMD Chips Guard Against Trojan Horses, IDG News Service [HTML] http://www.pcworld.com/news/article/0%2Caid%2C114328%2C00.asp , (Jan. 14, 2004)

16 Solar Designer, Linux kernel patch from the Openwall Project (Non-Executable User Stack), [HTML] http://www.openwall.com/

17 PaX Team , Documentation for the PaX project, [HTML] http://pax.grsecurity.net/docs/index.html

18 Ingo, Exec Shield, new Linux security feature, [HTML] http://redhat.com/~mingo/exec-shield/

19 S. Bhatkar, D. C. DuVarney, and R. Sekar, Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits, [HTML] Proceedings of the 12th USENIX Security Symposium, Washington, D.C. (Aug. 2003)

20 J. Etoh., GCC extension for protecting applications from stack-smashing attacks, [HTML] http://www.trl.ibm.com/projects/security/ssp/ (Jun. 2000)

21 L. Torvalds, Re: [PATCH] [SECURITY] suid procs exec'd with bad 0,1,2 fds, NEWS Archive, [HTML] http://old.lwn.net/1998/0806/a/linus-noexec.html (Aug. 1998)

22 Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle, PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities, // Proceedings of the 12th USENIX Security Symposium, Washington DC (Aug. 4-8, 2003)

23 Z. Shao, Q. Zhuge, Y. He, E. H.-M. Sha, Defending Embedded Systems Against Buffer Overflow via Hardware/Software, // Proceeding of the 20th Annual Computer Security Applications Conference, Tucson, Arizona (Dec. 6-10, 2004)

24 N. Tuck, B. Calder, and G. Varghese, Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow, // Proceedings of the 37th International Symposium on Microarchitecture, (Dec. 2004)

25 Vendicator, Stack Shield technical info file v0.7, [HTML] http://www.angelfire.com/sk/stackshield/index.html

26 J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, Architecture Support for Defending Against Buffer Overflow Attacks // Workshop on Evaluating and Architecting Systems for Dependability (Oct. 2002)

27 J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, A Processor Architecture Defense against Buffer Overflow Attacks, // Proceedings of the IEEE International Conference on Information Technology: Research and Education (ITRE 2003), pp. 243-250, (Aug, 2003)

28 H. Ozdoganoglu, T.N. Vijaykumar, C.E. Brodley, A. Jalote, and B. A. Kuperman, SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return

Address, (Nov.22, 2003)

29 A. Baratloo, N. Singh and T. Tsai, Transparent Run-Time Defense Against Stack Smashing Attacks // Proceedings of the USENIX Annual Technical Conference (2000)

30 M.S. M. Frantzen, StackGhost: Hardware facilitated stack protection // Proceedings of the 10th USENIX Security Symposium (Aug. 200)

31 T. Chiueh, F. Hsu, RAD: A Compile-Time Solution to Buffer Overflow Attacks, // International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, USA (Apr. 2001)

32 M. Prasad, and T. Chiueh, A Binary Rewriting Defense against Stack based Buffer Overflow Attacks, // Usenix Annual Technical Conference, General Track, San Antonio, TX (Jun. 2003)

33 Jones, R. W. M. and Kelly, P.H.J. Backwards-compatible bounds checking for arrays and pointers in C programs, // Third International Workshop on Automated Debugging, Linkoping University Electronic Press (1997)

34 Rational PurifyPlus, [HTML] http://www-3.ibm.com/software/awdtools/purifyplus/

35 BoundsChecker, [HTML] http://www.compuware.com/products/devpartner/bounds.htm

36 Robert P. Colwell, et al, Instruction Sets and Beyond: Computers, Complexity and Controversy // IEEE Computer, 18(9) (1985)

37 Organick, Elliott, A programmer's View of the Intel 432 System, McGraw-Hill, New York, N.Y. (1983)

38 U.S. Department of Energy Computer incident Advisory Capability, O-130: Perl and ActivePerl win32_stat Buffer Overflow, [HTML] http://www.ciac.org/ciac/bulletins/o-130.shtml

(Apr. 29, 2004)

39 Sun Alert Notification, Document ID 57643: Netscape NSS Library Vulnerability Affects Sun Java Enterprise System, [HTML] http://sunsolve.sun.com/search/document.do?assetkey=1-26-57643-1 (Sep. 16, 2004)

40 D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond // In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA (1996)

41 Intel Corporation, LaGrande Technology Architectural Overview (Sept, 2003)

42 Microsoft Corporation, The Next-Generation Secure Computing Base: An Overview, [HTML] http://www.microsoft.com/resources/ngscb/default.mspx

43 R. MacDonald, S. W. Smith, J. Marchesini, O. Wild, Bear: An Open-Source Virtual Secure Coprocessor based on TCPA, Technical Report TR2003-471, Department of Computer Science, Dartmouth College (Aug. 2003)

44 Trusted Computing Platform Alliance, TCPA IT White paper, [PDF] http://www.trustedcomputing.org

45 ARM, TrustZone Technology, [HTML] http://www.arm.com/products/CPUs/archtrustzone.html

46 G. S. Kc, A. D. Keromytis, and V. Prevelakis, Countering Code-Injection Attacks With Instruction-Set Randomization // Proceeding of the 10th ACM Conference on Computer and Communications Security

47 D. Kirovski, M. Drinic, and M. Potkonjak., Enabling Trusted Software Integrity // Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (2002)

48 G. C. Necula, S. McPeak, and W. Weimer, CCured: Type-Safe Retrofitting to Legacy Code // The Proceedings of the Principles of Programming Languages (2002)

49 John Viega, J.T. Bloch, Yoshi Kohno, Gary McGraw, ITS4: A Static Vulnerability Scanner for C and C++ Code // Proceeding of the 16th Annual Computer Security Applications Conference, New Orleans, Louisiana (Dec. 2000)

50 Flawfinder [HTML] http://www.dwheeler.com/flawfinder/

51 RATS,[HTML] http://www.securesw.com/rats/

52 D. Evans and D. Larochelle, Improving Security Using Extensible Lightweight Static Analysis // IEEE Software (Jan.-Feb. 2002)

53 E. Haugh, M. Bishop, Testing C Programs for Buffer Overflow Vulnerabilities // Proceedings of the 2003 Symposium on Networked and Distributed System Security (SNDSS 2003) (Feb. 2003)

54 D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities // Proceeding of the 10th USENIX Security Symposium (Aug. 2001)

55 C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman, FormatGuard: Automatic Protection From printf Format String Vulnerabilites // Proceedings for USENIX Security Symposium, Washington DC (Aug. 2001)

56 U. Shankar, K.Talway, J.S. Foster, and D. Wagner, Detecting Format String Vulnerabilities with Type Qualifiers // Proceedings of the 10th USENIX Security Symposium, pp. 201-216 (Aug. 2001)

57 J. Wilander and M. Kamkar, A Comparison of Publicly Available tools for Static intrusion

Prevention // Proceedings for7th Nordic Workshop on Secure IT systems, Karlstad, Sweden (2002)

58 J. Wilander and M. Kamkar, A Comparison of Publicly Available tools for Dynamic intrusion Prevention // 10th Network and Distributed system Security Symposium (NDSS) (2003)

59 Bulba and Kil3e, Bypassing stackguard and stackshield // Phrack Magazine, 10(56) May 2002.

60 T. Newsham, BugTraq Archive: Re: StackGuard: Automatic Protection From Stacksmashing Attacks, [HTML] http://www.securityfocus.com (Dec. 19, 1997)

61 D. Litchfield, Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server, NGSSoftware Ltd, [PDF] http://www.nextgenss.com (Sep. 2003)

62 F. Chang, A. Itzkovitz, V. Karamcheti, User-level Resource-constrained Sandboxing, // Proceedings for USENIX Windows System Symposium (Aug. 2000)

63 D. S. Peterson, M. Bishop, R. pandey, Flexible Containment Mechanism for Executing Untrusted Code, Proceedings of the 11th USENIX UNIX Security Symposium, pp. 207-225 (Aug. 2002)

64 M. Bishop, Computer Security, Addison-Wesley, (Dec. 2002)

65 SimpleScalar, [HTML] http://www.simplescalar.com/

66 Webopedia Computer Dictionary, What is buffer overflow?, [HTML] http://webopedia.com/TERM/b/buffer_overflow.html