переполнение буфера — активные средства защиты

крис касперски

умные в споре ищут истину,
глупцы — выясняют, кто умнее
народное

на рынке имеется множество средств (как коммерческих, так и бесплатных), обещающих решить проблему переполняющихся буферов раз и навсегда, но хакеры ломают широко разрекламированные защитные комплексы один за другим. почему? давайте заглянем под капот Stack-Guard, Stack-Shield, Pro-Police и Microsoft Visual Studio .NET, сравнив заявленные возможности с реальными

введение

Ошибки переполнения вездесущи — это факт. Буквально каждые несколько дней обнаруживается новая дыра, а сколько дыр остаются необнаруженными — приходится только гадать. Как с ними борются? Арсенал имеющихся средств довольно разнообразен и простирается от аппаратных защит типа NX/XD битов[1], до статических анализаторов наподобие Spilnt.

В последнее время в обиход вошел термин "secure programming" и вышло множество книг по безопасности, настоятельно рекомендующих использовать динамические средства защиты типа Stack-Guard, внедряющие в компилируемую программу дополнительный код, проверяющий целостность адреса возврата перед выходом из функции и предпринимающий другие действия, затрудняющие атаку.

Расплатой за "безопасность" становится снижение производительности (впрочем, довольно незначительное) и необходимость перекомпиляции всего кода. Но это только внешняя сторона проблемы. Понадеявшись на широко разрекламированные защитные средства, разработчики расслабляются и… начинают строчить небрежный код, который Stack-Guard (Stack-Shield/Pro-Police) все равно "исправит". Но что именно он правит? Давайте задвинем рекламу в сторону и посмотрим на защиту глазами хакера, который ломится не в дверь (там замок), и не в окно (там — сигнализация), а проникает через никем не охраняемую вентиляционную/канализационную трубу или даже дымоход.

Все защитные механизмы, имеющиеся на рынке, спроектированы так, что дрожь берет. Сразу видно, что их создатели никогда не атаковали чужие системы, не писали shell-код и даже не общались с теми, кто всем этим занимается. Защита не только не останавливает атакующего, но в некоторых случаях даже упрощает атаку!

типы переполнения и типы защит

Существует множество типов ошибок переполнения, подробно рассмотренных в статье "ошибки переполнения буфера извне и изнутри как обобщенный опыт реальных атак"). Это: переполнение кучи (работающее как оператор POKE [запись значения в указанную ячейку памяти]), целочисленное переполнение, ошибки форматированного вывода (PEEK [чтение содержимого произвольной ячейки памяти] и POKE в одном лице) и переполнение локальных стековых буферов.

Стековое переполнение — не только не единственное, но даже не самое популярное. Оператор new языка Си++ размещает переменные в динамической памяти, поэтому, актуальность атак на кучу все растет, а к стеку интерес снижается. Ложка — хороша к обеду. После драки кулаками не машут. Защитники стека явно опоздали и теперь подтасовывают факты и разводят рекламу. Вот цитата из документации на Stack-Guard: "…emits programs hardened against "stack smashing" attacks. Stack smashing attacks are the most common form of

[1] в анонсе этой статьи была допущена грубая ошибка, превратившая биты в байты. а все потому, что мыщъх не в себе, а в ком-то другом. и с крышей у него проблемы. течет. а времени на капитальный ремонт нет. мыщъх сидит на ноотропах и ждет когда заберут на дурдом (в худшем случае). хвостом еще шевелит, но не всегда адекватно. короче, эта статья написана в слега измененном состоянии сознания, поэтому в ней… все может быть.

penetration attack. Programs that have been compiled with StackGuard are largely immune to stack smashing attack" ("*Stack-Guard закаляет программы против срыва стека – наиболее популярного типа удаленных атак. Программы, откомпилированные со Stack-Guard'ом приобретают крепкий иммунитет против этого*"). На самом деле, Stack-Guard всего лишь затрудняет подмену адреса возврата, то есть противодействует ***подклассу*** стековых атак, причем, противодействует весьма неумело. Тоже самое можно сказать и про остальные защиты, устанавливая которые мы не должны забывать, что они сражаются лишь с определенным типом атак, а на остальные просто не обращают внимания.

Поскольку, из рекламных проспектов (по недоразумению называемых "технической документацией") ничего конкретного выяснить невозможно, используем дизассемблер, достоверно показывающий, что делает та или иначе защита и чем она реально занимается.

Рисунок 1 по сути, стек это просто очень высокая труба

stack-guard

Первым, кто бросил вызов переполняющимся буферам, был Stack-Guard, представляющий собой заплатку для компиляторов gcc и eggs, распространяемую по лицензии GPL. Раньше его было можно скачать с www.cse.ogi.edu/DISC/projects/immunix/StackGuard или immunix.org, но сейчас эти ссылки мертвы, а проект заброшен. С тех пор как Immunix скупила Novell, Stack-Guard больше не поддерживается, во всяком случае найти какие бы то ни было упоминания о нем на официальном сайте мне не удалось. Исходный код сохранился только у "коллекционеров", как например: www.packetstormsecurity.org/UNIX/utilities/stackguard. Тут может возникнуть вопрос: "Если stack-guard Устарел и мертв, какой смысл его исследовать?". На самом деле, смысл есть. Stack-Guard — простейший защитный механизм, расковыряв который, мы сможем разобраться и со всеми остальными, тем более, что между ними наблюдается стройная эволюционная преемственность.

Возьмем следующую программу, с умышленно допущенной ошибкой переполнения, и посмотрим, сможет ли Stack-Guard ее защитить.

```
// дочерняя функция
f(char *msg)
{
        // объявляем локальные переменные
        int a; char buf[0x66];

        // копируем аргумент в буфер без контроля длины,
        // что на определенном этапе приводит к его переполнению
        a = *strcpy(buf, msg);

        // выходим из функции
        return a;
}

// материнская функция
int main(int argv, char **argc)
{
        int x; x = f(argc[1]);
}
```

Листинг 1 демонстрационная программа с переполняющийся буфером, которую мы будем защищать

Откомпилируем файл компилятором gcc с настойками по умолчанию (то есть без оптимизации) и загрузим полученный elf в дизассемблер, чтобы посмотреть как выглядит стандартный пролог/эпилог функции f().

```
function_prologue:
        push    ebp                     ; // сохраняем старый указатель карда
        mov     ebp, esp                ; // открываем новый кадр стека
        sub     esp, 98h                ; // резервируем место под локальные переменные

; // тело программы
        mov     eax, [ebp+arg_0]        ; // копируем аргумент в регистр eax
        mov     [esp+98h+var_94], eax   ; // кладем eax в стек
                                        ; // (выглядит как засылка eax в лок. переменную
                                        ; // но в действительности это такая передача
```

```
                                        ; // аргументов, необычно но компилятору удобно)

        lea     eax, [ebp+var_88]       ; // получаем указатель на лок. переменную var_88
        mov     [esp+98h+var_98], eax   ; // кладем его в стек
        call    _strcpy                 ; // вызываем _strcpy(&arg_0[0], &var_88[0])
        movsx   eax, byte ptr [eax]     ; // eax = *((signed char*) eax);
        mov     [ebp+var_C], eax        ; // копируем eax в локальную переменную var_C
        mov     eax, [ebp+var_C]        ; // копируем содержимое var_C в eax

function_epilogue:
        leave                           ; // mov esp, ebp/pop ebp
        retn                            ; // выходим в материнскую функцию
```

Листинг 2 дизассемблерный листинг исходной функции f с мыщъх'иными комментариями

Содержимое стека на момент вызова f() представляет конгломерат локальных переменных и служебных данных. На вершине стека лежит буфер, под ним располагается целочисленная переменная "a" (на самом деле, порядок размещения переменных не стандартизован и целиком зависит от воли компилятора, то есть может быть любым). За локальными переменными следует сохраненный регистр указателя карда стека (в x86 процессорах его роль обычно играет EBP), а за ним — адрес возврата и аргументы, переданные функции. Короче говоря, все это выглядит так:

```
[    buf    ] ; ← переполняющийся буфер
[     a     ] ; ← прочие локальные переменные
[    ebp    ] ; ← сохраненный указатель кадра
[  retaddr  ] ; ← адрес возврата в материнскую функцию
[   arg 1   ] ; ← аргументы, переданные функции
[ --------- ] ; ← \
[ --------- ] ; ←  +- кадр стека материнской функции
[ --------- ] ; ← /
```

Листинг 3 состояние стека на момент вызова функции f

Переполняющийся буфер может воздействовать на следующие объекты: а) локальные переменные, расположенные ниже его; б) сохраненный указатель карда стека; в) адрес возврата; г) аргументы, переданные функции; д) на кадр материнской функции. Все эти атаки подробно описаны в статье "ошибки переполнения буфера извне и изнутри как обобщенный опыт реальных атак", поэтому не будем повторяться, а лучше пропустим программу через StackGuard 1.0 и посмотрим, что это даст.

```
function_prologue:
        push 000AFF0Dh          ; // забрасываем canary word на стек
                                ; // (следовало это делать после сохранение ebp)

        push ebp                ; // сохраняем старый указатель карда в стеке
        mov ebp, esp            ; // открываем новый кадр
        sub esp, 98h            ; // резервируем место под локальные переменные

; // тело функции
; // (точно такое же, как и в прошлый раз)

function_epilogue:
        leave                   ; // закрываем кадр стека
        cmp esp,AFF0Dh          ; // проверяем целостность canary word
        jne canary_changed      ; // если canary изменено, прыгаем на canary_changed
        add esp,4               ; // удаляем canary из стека
        ret                     ; // возвращаемся в материнскую процедуру

canary_changed:                 ; // завершаем выполнение программы
        call __canary_death_handler
        jmp .                   ; // если завершить не удалось — зациклившемся
```

Листинг 4 дизассемблерный листинг функции f(), защищенной Stack-Guard'ом (добавленные защитой строки выделены жирным шрифтом)

```
[    buf    ]
[     a     ]
```

```
[    ebp    ]
[ 000aff0dh ]
[  retaddr  ]
[   arg 1   ]
[ --------- ]
[ --------- ]
[ --------- ]
```

Листинг 5 состояние стека функции f() на момент завершения выполнения пролога и начала выполнения ее тела

После защиты StackGuard'ом перед адресом возврата располагается константа 000AFF0Dh (в терминологии StackGuard'a — **canary word**), целостность которой проверяется перед выходом из функции. Суть в том, что комбинацию символов, слагающие canary word – \x00\x0A\xFF\x0D, очень трудно "воспроизвести" с помощью строковых функций, поскольку в языке Си символ нуля трактуется как "конец строки". Функция gets – одна из тех немногих, что обрабатывает ноль как обыкновенный символ, поскольку в качестве завершителя строки использует символ "возврата каретки".

При работе с ASCIIZ-строками "подделать" canary word невозможно! Адрес возврата можно считать надежно защищенным. Ведь, чтобы "дотянуться" до него, переполняющемуся буферу необходимо пересечь (и затереть) canary word! Разработчики торжествуют, а хакеры режут себя вены вдоль сосуда. Или… все-таки нет? Начнем с того, что на Unicode все эти ограничения не распространяются и canary word подделывается без труда (кстати говоря, пилотная версия Stack Guard'a в качестве сторожевого слова использовала 00000000h, что в Unicode уже не воспроизводится, но может быть введено с помощью функции gets, которая сегодня практически никем и нигде не используется). К тому же, приложения, обрабатывающие двоичные данные функциями типа memcpy, так же остаются беззащитными.

Локальные переменные и указатель кадра стека вообще никак не защищены и могут быть беспрепятственно атакованы. Если среди этих переменных присутствует хотя бы один указатель на функцию, вызываемую после переполнения, хакер сможет подменить его адрес, передавая управление на свой shell-код. Конструкция типа "int *x; int a; … x = a;", которая к числу экзотических никак не относится, позволяет атакующему модифицировать любые указатели на функции, в том числе и адрес возврата и защита canary word'ом уже не срабатывает, поскольку сторожевое слово остается в неприкосновенности. Образно говоря, это как положить _перед_ шматком сала грозный капкан. Тот кто идет напрямую (классическое последовательное переполнение) попадет в него прежде, чем успеет полакомится. Но если десантироваться прямо на сало путем воздействия на переменные-указатели — капкан отдыхает (правда, в этом случае необходимо знать точное положение вершины стека на момент атаки, что не всегда возможно, поэтому хакеры предпочитают модифицировать таблицу импорта в Windows, а в UNIX – секцию got).

Рассмотрим самый сложный случай, когда никаких переменных в нашем распоряжении нет, а есть только сохраненный регистр кадра стека, который мы и будем атаковать. Фатальной ошибкой StackGuard'a явилось то, что он не учел "побочных эффектов" инструкции leave, которая работает так: `mov esp, ebp/pop ebp`, позволяя хакеру воздействовать на кадр материнской функции. Если в каком-то месте стека или кучи атакующему удастся "сложить" конструкцию `000AFF0Dh &shell-code`, (что в переводе на русский звучит как: canary-word за которым следует указатель на shell-код), ему остается всего лишь подменить сохраненный EBP на адрес "своего" canary-word. Тогда при выходе из материнской функции управление будет передано на shell-код! Атаки этого типа называются ret2ret и давно описаны в хакерской литературе, однако, какого-либо практического приложения они так и не получили, поскольку, в оптимизированных эпилогах (ключ -O2) вместо инструкции leave компилятор использует более быстродействующую конструкцию `add esp,x/pop ebp`, и побочный эффект воздействия на ESP исчезает. В оптимизированном эпилоге, хакер может воздействовать только на стековый кадр материнской функции, "подсовывая" ей те значения локальных переменных, которые он захочет. Для успешной реализации атаки этого, обычно, оказывается вполне достаточно.

В версии 2.0 защита адреса возврата была как бы усилена — в нем появился случайный canary word, хранящийся в read-only памяти и "шифрующий" адрес возврата по XOR. Угадать 32-битный canary word — нереально, но это и не нужно! Достаточно подсунуть заведомо ложное значение. Тогда, убедившись, что стек переполнен и хакеры хакерствуют как крысы в

амбаре, Stack-Guard передаст управление функции __canary_death_handler, которая завершает выполнение программы, устраивая настоящий DoS. Но лучше DoS, чем захват управления!

Рисунок 2 адрес возврата поXOR'ый случайным canary

Весь фокус в том, что указатель на __canary_death_handler размещается в глобальной таблице смещений — GOT и может быть атакован путем воздействия на локальные переменные через уязвимый указатель кадра стека. Если такие переменные действительно есть (а куда бы они подевались?), хакер просто перенаправляет __canary_death_handler на свой shell-код!

В последующих версиях Stack-Guard'a canary world "переехал" на одну позицию вверх, взяв под свою защиту и указатель кадра, однако, дальнейшего развития проект не получил и постепенно сдулся.

Рисунок 3 stack-guard – спящий стражник

Microsoft Visual Studio .NET

Озабоченная последними хакерскими атаками, Microsoft реализовала в своем новом компиляторе Visual Studio .NET (бывший Visual C++) некоторую разновидность Stack-Guard'a в далеко не лучшей его "инаугурации". Никогда не разрабатывающая собственных продуктов, а только "ворующая" уже готовые (авторитетный товарищ Берзуков в своей софт-панораме об этом только и говорит, сходите на www.softpanorama.org/Bulletin/News/Archive/news078.txt, почитайте — там много интересного), Microsoft, как это часто и бывает, сама не поняла, что стащила и у кого. Ладно, все это лирика. Перейдем к фактам.

При компиляции с ключом /GS компилятор добавляет в код **security cookie** — так москали кличут пыво, то есть, так в терминологии Microsoft называется случайный 32-битный canary word, хранящийся в writable-памяти и инспектируемый функцией check_canary при выходе из функции:

```
function _prologue:
        push ebp                ; // сохраняем прежний указатель кадра
        mov ebp, esp            ; // открываем новый кадр
        sub esp, 9Ch            ; // резервируем место для лок. переменных и canary
        push edx                ; \
        push esi                ;  + - сохраняем регистры которые будут изменены
        push edi                ; /

        mov eax, [canary]       ; // копируем глобальный canary в eax
        xor eax, [esp+10h]      ; // XOR'им адрес возврата с canary
        mov [ebp-10h],eax       ; // кладем результат на стек, защищая указатель кадра

; // тело функции
; (не совсем такое же, как и в прошлый раз,
;  но различия между компиляторами к делу не относятся)

function_epilogue:
        mov ecx, [epb-10h]      ; // копируем поXOR'ый canary в ecx
        xor ecx, [ebp+10h]      ; // XOR'им адрес возврата и кладем его ecx
        call check_canary       ; // вызываем функцию проверки canary

        pop edi                 ; \
        pop esi                 ;  + - восстанавливаем регистры
        pop ebx                 ; /
        mov esp, ebp            ; // закрываем кадры стека небезопасным путем
        pop ebp                 ; // (Microsoft повторяет ошибку Stack Guard'a)
        ret                     ; // выходим в материнскую функцию

check_canary:                   ; // функция проверки canary
        cmp ecx, [canary]       ; // сравниваем переданный ecx с глобальным canary
        jnz canary_changed      ; // если не совпадают - завершаем программу
        ret                     ; // все ок, продолжаем выполнение программы
```

Листинг 6 дизассемблерный листинг функции f(), откомпилированной Microsoft .NET с ключом /GS (добавленные защитой строки выделены жирным шрифтом)

Canary word защищает не только адрес возврата, но и кадр, что очень хорошо, правда в оптимизированном коде, генерируемый этим же самым компилятором, локальные переменные

адресуются непосредственно через ESP и дополнительный регистр им не нужен, поэтому, фактически защищается только один адрес возврата. Остальные переменные остаются незащищенными, что открывает простор для махинаций с указателями. В частности, хакер может перезаписать глобальную переменную canary своим значением — тогда его проверка пройдет нормально. Это даже ***упрощает*** (!) атаку: в незащищенной системе существует проблема ввода "запрещенных" символов, которую не всегда возможно обойти. Операция XOR позволяет генерировать любые символы! В частности, чтобы сформировать символ нуля, достаточно положить в canary и зашифрованный адрес возврата два одинаковых символа. Как известно X XOR X = 0. Остальные символы генерируются аналогичным способом.

Самое интересное, что Microsoft переняла ошибку ранних версий Stack-Guard, причем даже не его ошибку, а особенность поведения компилятора GCC, позволяющую атакующему воздействовать на регистр ESP через модификацию указателя кадра стека. Microsoft Visual C++ 6.0 закрывал кадр стека безопасной конструкций ADD ESP,XXX, а .NET вместо этого использует MOV ESP, EBP. И хотя указатель кадра защищен canary word, это еще не повод ослаблять защиту! Canary word генерируется не совсем случайным путем и угадать его с нескольких попыток вполне реально, ну а инструкция XOR позволит подделать любой символ. Короче говоря, если бы в Microsoft думали головой…

stack-shield

Несмотря на схожесть в названии со своим собратом, Shack-Shield действует совсем по другому принципу. Это еще одно расширение к gcc, последнюю версию которого можно скачать с http://www.angelfire.com/sk/stackshield, но иного типа. Если Stack-Guard реализован как патч к компилятору, "исправляющий" function_prologue и function_epilogue, то Stack-Shield "захватывает" ассемблерные файлы, сгенерированные компилятором (в UNIX-мире они имеют расширение .S), обрабатывает их, выплевывая защищенный ассемблерный файл, возвращаемый компилятору для окончательной трансляции в двоичный код. Такая схема дает Stack-Shiled'y намного большие возможности и автору этой статьи сразу же захотелось посмотреть как он ими воспользовался и можно ли его одолеть.

Рисунок 4 stack-shield – щит, защищающий стек

Соблазненный процессорными архитектурами с разнесенным стеком (один стек для хранения адресов возврата, другой — для локальный переменных), создатель Stack-Guard'a попытался "проэмулировать" на x86 нечто подобное. Для этой цели он использовал глобальный массив retarray на 256 адресов: эпилог копирует текущий адрес на вершину массива, определяемую указателем retprt, а пролог "стягивает" этот адрес с вершины и передает ему управление. Эта эмуляция далека от идеала, но сохраненный в стеке адрес возврата в ней вообще не используется и выполнение программы продолжится даже после того, как он будет затерт, что предотвращает DoS (впрочем, поскольку локальные переменные искажены, программа все равно рухнет).

```
function_prologue:
        push eax                ; // сохраняем регистры, которые изменяет Stack-Shied
        push edx
        mov eax, offset retpt   ; // копируем в eax смещение указателя массива retpt
        cmp rettop, eax         ; // смотрим - есть ли еще место?
        jbe .LSHIELDPROLOG      ; // если места нет,отказываемся от записи нового адреса
        mov edx, [esp+8]        ; // заносим в edx адрес возврата со стека
        mov [eax], edx          ; // сохраняем его в массиве адресов возврата

.LSHIELDPROLOG:
        add [retptr],4          ; // увеличиваем указатель массива возвратов
                                ; // на первый взгляд это явный баг,
                                ; // но на самом деле - оптимизация!

        pop edx                 ; // восстанавливаем регистры назад
        pop eax                 ;

        push ebp                ; // сохраняем старый указатель карда стека
        mov ebp, esp            ; // открываем новый кадр
        sub esp, 98h            ; // резервируем место под локальные переменные

; // тело функции
; // (такое же как в случае с Stack-Guard)
```

```
function_epilogue:
        leave                   ; // закрываем кадр стека небезопасным путем
        push eax                ; // сохраняем регистры
        push edx
        add [retptr], -4        ; // уменьшаем указатель массива возвратов
        mov eax, offset retptr  ; // заносим в eax смещение массива возвратов
        cmp eax, rettop         ; // как на счет свободного места?
        jbe .LSHIELDEPILOG      ; // если места нет, значит и выталкивать нечего
        mov edx, [eax]          ; // снимаем сохраненный адрес со стека возвратов
        mov [esp+8],edx         ; // восстанавливаем стековый адрес не проверяя его

.LSHIELDEPILOG:
        pop edx                 ; // восстанавливаем регистры
        pop eax                 ;
        ret                     ; // выходим в материнскую функцию
```

Листинг 7 дизассемблерный листинг функции f(), защищенной Stack-Shield'ом с настройками по умолчанию (добавленные защитой строки выделены жирным шрифтом)

При компиляции с ключом -d, Stack-Shiled вставляет дополнительную проверку, сравнивая адреса возврата на стеке и retarray'e. В случае расхождения вызывается функция SYS_exit, завершающая программу в аварийном режиме.

Ключи -r и -g задействуют механизм "Ret Range Checking", проверяющий границы адресов возврата и останавливающий программу, если они выходят за пределы некоторой заранее заданной величины (т. е. находятся в куче или стеке). Таким образом, даже если хакер перезапишет retarray (а он находится в записываемой области памяти), подсунуть указатель на shell-код ему уже не удастся, правда, он может беспрепятственно вызывать функции библиотеки libc, передавая им любые аргументы (атака типа return-to-libc).

```
function_epilogue:
        leave                   ; // закрываем кадр стека небезопасным путем
        cmp [esp], offset shielddatabase
                                ; // ^ проверяем границы адреса возврата
        jbe .LSHIELDRETRANGE    ; // если все ок, то прыгаем на ret

        movl eax,1              ; // если мы здесь, то адрес возврата вышел
        movl ebx,-1             ; // за допустимые пределы, возможно он был изменен
        int 80h                 ; // завершаем выполнение программы

.LSHIELDRETRANGE:
        ret                     ; // возвращаемся в материнскую процедуру
```

Листинг 8 дизассемблерный код, раскрывающий сущность механизма Ret Range Checking

Усилилась и защита локальных переменных. Теперь перед вызовом функции по указателю, Stack-Shield убеждается, что она находится в пределах сегмента кода (см. листинг 9):

```
                                ; // в eax находится указатель на функцию
        cmp eax, offset shielddatabase
                                ; // проверяем границы указателя

        jbe .LSHIELDCALL        ; // если указатель в границах, прыгаем на вызов функции

        mov eax,1               ; // указатель на функцию выходит за допустимые границы
        movl ebx,-1             ; // возможно, он был хакнут
        int 80h                 ; // завершаем выполнение программы

.LSHIELDCALL:
        call [eax]              ; // вызываем функцию по указателю
```

Листинг 9 дизассемблерный код, показывающий как Stack-Shield контролирует указатели на функции

Контроль за указателями на функции препятствует непосредственной передаче управления на shell-код, но не мешает хакеру использовать функции стандартной библиотеки libc и функции самой уязвимой программы. Указатели на данные так же остаются незащищенными. Кроме того, при исчерпании массива адреса возвратов (что при глубокой вложенности функций имеет место быть), он автоматически переход в "обычный" режим, в котором проверяет только границы адресов возврата, но не сами адреса. Хорошая новость, нечего сказать!

pro-police

Протектор Pro-Police, зародившийся в недрах японского отделения IBM (http://www.research.ibm.com/trl/projects/security/ssp/), — это, без преувеличения, самый сложный и самый совершенный механизм, реализующий модель безопасного стека (Safe Stack Usage Model), который действительно защищает, а не разводит пропаганду, чтобы выбить очередной грант. Сражение с такой защитой любой самурай почтет за честь.

Pro-police зарывается намного глубже, чем Stack-Guard и работает на уровне RTL. Это не библиотека времени исполнения, это — промежуточный системно-независимый язык, генерируемый компилятором gcc и расшифровываемый как register transfer language. Абстрагирование от оборудования существенно упрощает портирование и pro-police поддерживает практически все современные платформы: x86, powerpc, alpha, sparc, mips, vax, m68k, amd64.

Самая главная инновация — переупорядочивание локальных переменных. Pro-police разбивает переменные на две группы: массивы и все остальные. На вершину карда стека попадают обычные, скалярные, переменные. Массивы идут за ними. Переполняющиеся буфера могут воздействовать друг на друга, но до указателей уже не достать, во всяком случае не таким простым путем.

Адрес возврата и указатель кадра защищены сторожевой константой guard, генерируемой произвольном образом. Это все тоже canary word, только в обличии новой терминологии.

```
foo()
{
        char *p;                        // локальная переменная-указатель
        char buf[128];                  // локальный буфер

        gets (buf);                     // функция, которая этот буфер и переполняет
}
```

Листинг 10 псевдокод уязвимой функции до защиты Pro-Police

```
Int32   random_number;                  // глобальный canary, генерируемый случ. образом
foo ()
{
        volatile int32 guard;           // локальная копия canary, охраняющая кадр
        char buf[128];                  // буфер идет перед всеми локальными переменными!
        char *p;                        // локальная переменная-указатель

        guard = random_number;          // копируем глобальный canary в лок. переменную

        gets (buf);                     // вызываем уязвимую функцию

        if (guard != random_number)     /* program halts */
}
```

Листинг 11 псевдокод функции, защищенной Pro-Police (добавленные защитой строки выделены жирным шрифтом)

Состояние стека на момент вызова функции f из листинга 1 под Pro-Police выглядит так:

```
[      p      ]
[    buf      ]
[   guard     ]
[    ebp      ]
[  retaddr    ]
[   arg 1     ]
[ ---------   ]
[ ---------   ]
[ ---------   ]
```

Листинг 12 ссостояние стека функции foo() на момент завершения выполнения пролога, обратите внимание, что при переполнении буфера buf затирания локальных переменных уже не происходит!

Сравните это с листингом 5. Разница незначительная, но принципиальная! По соображениям производительности, Pro-Police внедряет защиту адреса возврата только функции содержащие буфера, которые потенциально могут быть переполнены. То есть, Pro-Police совмещает в себе защитный механизм с системой аудита кода!

Рисунок 5 безопасная модель стека Pro-Police

Pro-police предусматривает даже такую неочевидную ситуацию, как подмену указателей, переданных в качестве аргументов и надежно защищает их. В прологе аргументы копируются в промежуточные переменные, расположенные "над" переполняющимся буфером, а не "под" ним (где находятся оригинальные аргументы). В дальнейшем, все обращения к аргументам осуществляются через промежуточные переменные следующим образом:

```
foo (int a, void (*fn)())
{
        char buf[128];                  // локальный буфер

        gets (buf);                     // функция, переполняющая буфер
        (*fn)();                        // вызов функции, по указателю переданному
                                        // в качестве аргумента и затираемому
                                        // при переполнении
}
```

Листинг 13 псевдокод уязвимой функции, вызывающей функцию по указателю передаваемому в качестве аргумента, до защиты Pro-Police

```
Int32   random_number;                  // глобальный canary, генерируемый случ. образом
foo (int a, void (*fn)())               // уязвимый аргумент-указатель
{
        volatile int32 guard;           // локальная копия canary, охраняющая кадр
        char buf[128];                  // буфер идет перед переменными, но после аргум.
        (void *safefn)() = fn;          // копируем аргумент во временную переменную
        guard = random_number;          // копируем глобальный canary в лок. переменную

        gets (buf);                     // вызываем уязвимую функцию

        (*safefn)();                    // вызываем функцию по скопированному указателю
        if (guard != random_number)     /* program halts */
}
```

Листинг 14 псевдокод функции, защищенной Pro-Police (добавленные строки выделены жирным шрифтом)

При всей надежности Pro-Police, отсутствие сторожевых слов между массивами, делает атаку по-прежнему возможной, поскольку затирание нижеследующих массивов порождает целый каскад вторичных переполнений (особенно целочисленных), да и массивы из указателей не такая уж большая редкость. Тем не менее такая проверка (кстати говоря, обещанная в следующих версиях Pro-Police) приведет к существенному падению производительности, что явно пойдет не на пользу ее популярности.

	stack-guard	.NET	stack-shield	pro-police
защищает адрес возврата	да	да	частично	да
защищает указатель кадра	нет	да	нет	да
защищает локальные переменные	нет	нет	частично	да
защищает аргументы	нет	нет	нет	да
защищает массивы	нет	нет	нет	нет
canary word случаен	нет	частично	—	да
защищает canary word от перезаписи	да	нет	—	да

Таблица 1 сводная таблица различных защитных методов

заключение

Так все-таки можно защититься от переполняющихся буферов или нет? Pro-Police отсекает большое количество атак, но… все это атаки на стек, а помимо стека у нас еще есть

целочисленное переполнение, спецификаторы и куча, которые Pro-Police даже не пытается охранять, поскольку они находятся вне его "департамента". Это не упрек, а скорее констатация факта.

Личное наблюдение — прочитав несколько популярный статей и установив могучий Pro-Police, большинство знакомых мне программистов упускают из виду, что необходимо установить что-то еще. Безопасное программирование требует целого комплекса совокупных мер, жестоко карая за малейшие ошибки.

Использовать Pro-Police безусловно стоит, равно как и компилировать программы с ключом /GS, однако, необходимо помнить, что эта мера отнюдь не гарантирует защищенности, а всего лишь уменьшает вероятность атаки.