Можно ли защититься от переполнения буферов?

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.

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 до статических анализаторов наподобие Spilnt.

В последнее время в обиход вошел термин «secure programming» и издано 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» становятся 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 [1]. Это:

- **переполнение кучи** (работающее как оператор POKE – запись значения в указанную ячейку памяти);
- **целочисленное переполнение, ошибки форматированного вывода** (PEEK – чтение содержимого произвольной ячейки памяти) POKE в одном лице);
- **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 дорога 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 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 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. То же 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.

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 http://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: http://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.

Листинг 1. Демонстрационная программа с переполняющимся буфером, которую мы будем защищать

```
// 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]);
}
```

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

Листинг 2. Дизассемблерный листинг исходной функции f с моими комментариями

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

; // тело программы
      ; // копируем аргумент в регистр eax
      mov       eax, [ebp+arg_0]
      ; // кладем eax в стек (выглядит как засылка eax
      ; // в локальную переменную но в действительности –
      ; // это такая передача аргументов, необычно,
      ; // но компилятору удобно)
      mov       [esp+98h+var_94], eax

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

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

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 кадра стека (в x86 процессорах его роль обычно играет EBP), а за ним – адрес возврата и аргументы, переданные функции. Короче говоря, все это выглядит так:

Листинг 3. Состояние стека на момент вызова функции f

```
[    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
[ --------- ] ; <-- /
```

Переполняющийся буфер может воздействовать на следующие объекты:

- локальные переменные, расположенные ниже его;
- сохраненный указатель кадра стека;
- адрес возврата;
- аргументы, переданные функции;
- на кадр материнской функции.

Все эти атаки подробно описаны в моей статье [1], поэтому не будем повторяться, а лучше пропустим программу через Stack-Guard 1.0 и посмотрим, что это даст.

Листинг 4. Дизассемблерный листинг функции f(), защищенной Stack-Guard (добавленные защитой строки выделены красным шрифтом)

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

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

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

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

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

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

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

После защиты Stack-Guard перед адресом возврата располагается константа 000AFF0Dh (в терминологии Stack-Guard – canary word), целостность которой проверяется перед выходом из функции. Суть в том, что комбинацию символов, слагающих 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 стреляются. 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 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, также 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 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 полакомиться. 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 Stack-Guard 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.

Рисунок 1. Адрес возврата по XOR случайным canary

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!

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 canary world «переехал» на одну позицию вверх, взяв под свою защиту и указатель кадра, однако дальнейшего развития проект не получил и постепенно сдулся.

Microsoft Visual Studio .NET

Озабоченная последними хакерскими атаками, Microsoft реализовала в своем новом компиляторе Visual Studio .NET (бывший Visual C++) некоторую разновидность Stack-Guard 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 в терминологии Microsoft называется случайный 32-битный canary word, хранящийся в writable-памяти и инспектируемый функцией check_canary при выходе из функции:

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

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

```
                        :       которые будут изменены
        push esi
        push edi            ; /

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

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

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

        pop edi             ; \
        pop esi             ;  + - восстанавливаем регистры
        pop ebx             ; /

        ; // закрываем кадры стека небезопасным путем
        mov esp, ebp

        ; // (Microsoft повторяет ошибку Stack Guard)
        pop ebp
        ; // выходим в материнскую функцию
        ret

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

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 намного большие возможности, и мне сразу же захотелось посмотреть, как он ими воспользовался и можно ли его одолеть.

Соблазненный процессорными архитектурами с разнесенным стеком (один стек для хранения адресов возврата, другой – для локальный переменных), создатель Stack-Guard 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).

Листинг 7. Дизассемблерный листинг функции f(), защищенной Stack-Shield с настройками по умолчанию (добавленные защитой строки выделены красным шрифтом)

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

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

        ; // восстанавливаем регистры назад
        pop edx
        pop eax             ;

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

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

function_epilogue:
        ; // закрываем кадр стека небезопасным путем
        leave
```

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

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

При компиляции с ключом -d, Stack-Shiled вставляет дополнительную проверку, сравнивая адреса возврата на стеке и retarray. 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).

Листинг 8. Дизассемблерный код, раскрывающий сущность механизма Ret Range Checking

```
function epilogue:
        ; // закрываем кадр стека небезопасным путем
        leave
        cmp [esp], offset shielddatabase

        ; // ^ проверяем границы адреса возврата
        ; // если все ок, то переходим на ret
        jbe .LSHIELDRETRANGE

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

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

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

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

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

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

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

.LSHIELDCALL:
```

Рисунок 2. Безопасная модель стека Pro-Police

```
        call [eax]          ; // вызываем функцию по указателю
```

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 также остаются незащищенными. Кроме того, при исчерпании массива адреса возвратов (что при глубокой вложенности функций имеет место быть) он автоматически переходит 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 кадра стека попадают обычные скалярные переменные. Массивы идут за ними. Переполняющиеся буферы 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.

Листинг 10. Псевдокод уязвимой функции до защиты Pro-Police

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

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

Сводная таблица различных защитных методов

	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

Листинг 11. Псевдокод функции, защищенной Pro-Police (добавленные защитой строки выделены красным шрифтом)

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

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

      gets (buf);        // вызываем уязвимую функцию

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

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

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

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

Сравните это с **листингом 5**. Разница незначительная, но принципиальная! По соображениям производительности, Pro-Police внедряет защиту адреса возврата только функции, содержащие буферы, которые потенциально могут быть переполнены. То есть Pro-Police совмещает в себе защитный механизм с системой аудита кода (**рис. 2**)!

Pro-Police предусматривает даже такую неочевидную ситуацию, как подмена 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:

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

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

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

Листинг 14. Псевдокод функции, защищенной Pro-Police (добавленные строки выделены красным шрифтом)

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

      gets (buf);        // вызываем уязвимую функцию

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

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.

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.

Литература:

1. Касперски К. Ошибки переполнения буфера извне и изнутри как обобщенный опыт реальных атак. – Журнал «Системный администратор» №3, 2004 г. – 64-72 с.