помощники компилятора

крис касперски ака мыщъх, no-email

это ваабще не наше и на оно, в принципе, на хрен не надо, но приходится пользовать ибо своего нет.

программистское

времена, когда компиляторы — компилировали, а программисты — программировали уже давно позади. современный компилятор это могучий инструмент, совмещающий в себе функциональность кухонного комбайна со стремительностью пикирующего бомбардировщика. когда же его возможностей оказывается недостаточно, на помощь приходит множество полезных (и не очень полезных) утилит, от изобилия которых начинает рябить в глазах. как же выбрать из всего этого хлама действительно нужное?

введение

Нет нужды говорить, что языки Си и Си++ не для прикладников. Этот [вам|совсем] не Паскаль, складывающий строки так же как и остальные типы данных, и не Ада с ее поддержкой динамических массивов и встроенным контролем границ. Идеологию Си хорошо выражают слова японского мультипликатора Миядзаки Хаяо — "*стоит ли использовать компьютер для того, что можно сделать руками?*" Обо всех проверках Си-программист должен заботиться самостоятельно и если хоть однажды он об этом забудет (или допустит небрежность), последствия в виде нестабильной работы, червей или утечек памяти не заставят себя ждать.

Казалось бы — не умеешь программировать на Си, выбирай другой язык, например, Яву или Фортран. Так ведь нет, не хотят! Упрекают создателей Си в кретинизме, но с него не слезают. Страустрапа вообще Дохлым Страусом обозвали. Попытки исправить язык, добавив в него, например, автоматический сборщик мусора, предпринимались неоднократно. В результате одного из таких проектов и родилась Ява, чем-то напоминающая корову с седлом, образ которой еще долго будет преследовать всех столкнувшихся с ней программистов. Медленно, неудобно и все равно небезопасно (заранее прощу прощения у всех поклонников этого, в общем-то неплохого языка, но против собственных ассоциаций не попрешь).

Дружелюбно настроенные программисты предлагают не трогать язык, оставив Си/Си++ таким, какой он есть (руки прочь! пасть порву!), но изменить компилятор, заставляя его внедрять проверочный код после каждой потенциально небезопасной операции. Еще предлагают переписать все стандартные библиотеки, научив их распознавать наиболее характерные ошибки распределения памяти… Расплатой за это становится значительное падение производительности, а это маст дай [а падение, как бы сказал Пятачок — это нехорошо].

Статические анализаторы все проверки выполняют до компиляции, тыкая программиста носом во все неблагонадежные месте [обращают внимание программиста на все неблагонадежные места], которые могут привести к проблемам. Пусть сам решает как их исправить. К сожалению, возможности статических анализаторов очень ограничены и многим ошибкам удается ускользнуть.

В общем, не ситуация, а одно разбитое корыто. С ошибками лучше всего справляться своей собственной головой, используя компилятор (и примочки к нему) как дополнительный уровень обороны. Сработает — хорошо, не сработает — ну и хрен с ним.

Вот о примочках к компиляторам мы и будем говорить. Их можно разделить на две категории — средства противохакерской защиты, предотвращающие переполнения буфера (а вместе с этим — и засылку shell-кода) и детекторы ошибок распределения памяти, удерживающие программу от входа вразнос. Все описываемые утилиты, во-первых, бесплатны, а, во-вторых, не зажимают исходные тесты.

хакеры под прицелом

Переполнения буфера чаще всего возникают не где-нибудь, а сосредоточены в строго определенных местах, которыми, как правило являются следующие функции: strcpy(), strcat(), gets(), sprintf(), семейство scanf()-функций, [v][f]printf(), [v]snprintf() и syslog(). В девяти из

десяти случаев передача управления на shell-код осуществляется путем подмены адреса возврата из функции. Остальные способы приходятся на модификацию индексов, указателей и прочих типов переменных. Причем, переполнение буфера как правило происходит последовательно, то есть затирается непрерывный регион памяти. Индексное переполнение, при котором затирается несколько ячеек далеко за концом буфера, носит эпизодический характер и большой опасности не представляет.

Это сужает круг "подозреваемых" и значительно упрощает задачу контроля за буферами. Существует множество утилит, предотвращающих (или, во всяком случае пытающихся предотвратить) переполнения. Вот только некоторые из них…

StackGuard

Вероятно, самый удачный и самый популярный анти-хакерский протектор, представляющий собой заплатку для GCC, модернизирующую машинный код пролога (function_prolog) и эпилога (function_prolog), вставляемый компилятором в начало и конец каждой функции. При входе в функцию, поверх адреса возврата устанавливается чувствительный индикатор (он же canary-word), неизбежно затираемый хакером при последовательном переполнении. Перед выходи из функции, canary-word сверяется с оригиналом, хранящемся в недосягаемом для хакера месте, и если его целостность окажется нарушенной, программа сообщит, что ее взломали и мирно отвалит, устроив себе настоящий DoS (отказ в обслуживании).

Для предотвращения подделки чувствительного детектора Stack Guard предпринимает целый ряд мер. Canary-word представляет собой комбинацию из трех термирующих спецсимволов (0x00000000L, CR, LF и FFh), которые большинство функций воспринимает как завершитель ввода, и случайной привязки, считываемой из устройства /dev/urandom или генерируемой на основе текущего времени, если /dev/urandom недоступно. Этот прием защищает лишь от последовательных переполнений (да и то не от всех), и бессилен против индексных.

При необходимости, Stack Guard может запрещать модификацию адреса возврата на время выполнения функции, что существенно усиливает защищенность, но вместе с тем, "роняет" производительность (canary-word быстродействия практически не сокращает). К тому же, для реализации данного механизма требуется определенная поддержка со стороны ядра, а в большинстве ядер ее нет.

#где взять: http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/

неисполняемый стек

Специальный падч от Solar'a Designer'a, встраивается в Линуховое ядро, делая стек неисполняемым. Переполняющиеся буфера по прежнему будут приводить к краху приложения, но непосредственная передача управления на shell-код становится невозможной, точнее возможной но очень трудно реализуемой (подробности в статье Defeating Solar Designer's Non-executable Stack Patch, выложенной на сервере http://www.insecure.org/sploits/non-executable.stack.problems.html).

Это не снижает производительности и не требует перекомпиляции существующих приложений, но на универсальное решение, увы, не тянет. Заплатки доступны только для старых версий ядер (2.0, 2.2, 2.4), да и всевозможных конфликтов предостаточно. Тем не менее, полностью отказываться от идеи неисполняемого стека все же не стоит.

#где взять: http://www.openwall.com/linux/

ITS4 Software Security Tool

Статический анализатор исходных текстов, нацеленный на поиск переполняющихся буферов и некоторых других ошибок. Отмечает вызовы потенциально опасных функций, таких например, как strcpy/memcpy, и выполняет поверхностный семантический анализ, пытаясь оценить насколько опасен такой код, а так же дает советы по его улучшению (в большинстве своем либо слишком очевидные, либо откровенно глупые). Поддерживает приплюснутый и обычный диалекты Си. Представляет собой утилиту командной строки, работающую как на Windows, так и на UNIX.

#где взять: http://www.cigital.com/its4/

Flawfinder

Простой статический анализатор исходных текстов, написанных на языках Си и Си++. Пытается обнаружить ошибки переполнения, но как же неумело он это делает! Вместо семантического анализа кода, нам предлагают простой шаблонный поиск. Flawfinder обращает внимание лишь на имя функции (strcpy, strcat и т.д.) и аргументы переданные ей (константная строка или указатель на буфер), оценивая потенциальную опасность в условных "хитах". Тем не менее полезен для получения общих представлений о программе, особенно чужой.
#где взять: http://www.dwheeler.com/flawfinder/

что-то с памятью моей стало

Проблемы с распределением памяти в основном относятся к чистому Си. В плюсах имеется множество механизмов для их решения. Конструкторы и деструкторы, перекрытие операторов, объекты с ограниченной зоной видимости — все это ликвидирует часть ошибок из серии "выделит память и забыл ее освободить". С другой стороны, более сложная семантика Си++ существенно затрудняет статический анализ, вынуждая прибегать к run-time контролю, осуществляемому непосредственно на стадии исполнения, что несколько снижает производительность.

Большой популярностью пользуются отладочные версии библиотек, осуществляющие жесткий контроль за динамической памятью (она же — куча) и обнаруживающие большое количество трудноуловимых ошибок, с которыми раньше приходилось справляться лишь многодневной отладкой без перерыва на сон и еду. По соображениям производительности обычно они используются лишь на стадии разработки и альфа-тестирования, а из финальной версии — исключаются.

CCured

Сишный протектор, защищающий программу от проблем с распределением памяти (выход за границы буфера, использование неинициализированных указателей и т. д.) и работающий по принципу source2source транслятора, "заглатывающего" сырой исходный текст и вставляющего дополнительные проверки в различных местах. То есть, вместо того чтобы исправить ошибки, он загоняет их подальше вглубь! Своеобразный предохранительный клапан, удерживающий программу от входа "вразнос" и предотвращающий ряд удаленных атак, основанных на передаче shell-кода. Но вот отказ в обслуживании злоумышленник устроить вполне может. К тому же, дополнительные проверки ощутимо замедляют быстродействие программы (от 10% до 60% в зависимости от качества исходного кода).

Маленькие программы транслируются автоматически, но в серьезных проектах над текстом, выданном CCrued'ом, приходится как следует поработать (то есть, CCrued не только "лечит" программы, по еще и портит!). Тем не менее, процесс "послеродовой" реабилитации защищенного листинга описан достаточно подробно, и разработчикам CCured'a удалось переварить исходные тексты sendmail'a, bind'a, openssl, Apache и других приложений, потратив на каждое из них по несколько дней. Тем не менее, run-time контроль, реализованный в CCured, намного надежнее статического анализа.
#где взять: http://manju.cs.berkeley.edu/ccured/

MEMWATCH

Набор отладочных функций для определения ошибок распределения памяти, поставляющийся в исходных текстах. Состоит из заголовочного файла MEMWATCH.H и ядра MEMWATCH.C, написанных на ANSI C, что обеспечивает совместимость со всеми "нормальными" компиляторами и платформами (разработчики заявляют от поддержке: PC-lint 7.0k, Microsoft Visual C++ (как 16-, так и 32-разрядные версии), Microsoft C для DOS, SAS C для Amiga 500, GCC и некоторых других). Поддержка Си++ находится в зачаточном состоянии.

Стандартные функции распределения памяти (malloc, realloc, free) оборачиваются в отладочную обертку, отслеживающую утечки памяти, двойное освобождение указателей, обращение к неинициализированным указателям и выход за пределы выделенного блока памяти. Создается некоторое количество сторожевых блоков, отлавливающих "дикие" указатели, обращающиеся к невыделенным областям памяти. Все обнаруженные ошибки записываются в журнал. Макросы ASSET и VERIFY заменяются их продвинутыми версиями,

вместо немедленного завершения сбойнувшей программы предлагающие пользователю стандартный набор действий: Abort-Retry-Ignore.

Платформенно-зависимая часть кода разработчиками не реализована и функции типа mwIsReadAddr/mwIsSafeAddr каждый вынужден дописывать самостоятельно. Другой серьезный недостаток — программа должна быть явным образом подготовлена для работы с MEMWATCH, что в ряде случаев неприемлемо. Многопоточность поддерживается лишь на рудиментом уровне и когда она будет реализована в полном объеме — неизвестно.
#где взять: http://www.linkdata.se/sourcecode.html

Dmalloc - Debug Malloc Library

Отладочная версия библиотеки для работы с памятью, замещающая собой штанные функции языка Си: malloc, realloc, calloc, free и др. Вносить изменения в исходный код приложения при этом не требуется (хотя при желании проверки распределения памяти можно осуществлять и явно).

Сервис, предоставляемый dmalloc'ом вполне стандартен сервис для утилит этого класса: утечки памяти, выход за границы буферов, статистика и логгинг с указанием номеров строк и имени файла. При включении проверок на каждую операцию ужасно тормозит, так что для работы потребуется по меньшей мере Pentium-4/Prescott.

Работает практически везде: AIX, BSD/OS, DG/UX, Free/Net/OpenBSD, GNU/Hurd, HPUX, Irix, Linux, MS-DOG, NeXT, OSF, SCO, Solaris, SunOS, Ultrix, Unixware, Windows, и даже под операционной системой Unicos на суперкомпьютере Cray T3E. Сложным образом конфигурируется и требует предварительной подготовки с докой в руках, что не есть плюс. Зато полноценно поддерживает много поточность, которой могут похвастаться далеко не все его конкруенты.
#где взять: http://dmalloc.com/

Checker

Еще одна отладочная библиотека, предлагающая свою реализацию функций malloc, realloc и free. Ругается всякий раз, когда free или realloc принимают указатель полученный не от malloc, отслеживает повторное освобождение уже освобожденных указателей и обращения к неинициализированным областям памяти. Откладывает реальное освобождение блоков памяти на некоторое время, в течении которого пристально следит: не происходит ли к ним каких-нибудь обращений. Содержит детектор "мусора", вызываемый либо из отладчика, либо непосредственно из самой исследуемой программы. В общем, простенько, но со вкусом. К тому же, практически без ущерба для производительности системы. Работает в паре с компилятором GNU, на других — не проверял.
#где взять: http://www.gnu.org/software/checker/checker.html

>>> простой макрос для обнаружения утечек памяти

```
#ifdef DEBUG
#define MALLOC(ptr,size) do { \
ptr = malloc (size); \
pthread_mutex_lock(&gMemMutex); \
gMemCounter++; \
pthread_mutex_unlock(&gMemMutex); \
}while (0)
#else
#define MALLOC(ptr,size) ptr = malloc (size)
#endif
```

Листинг 1 обертка вокруг malloc'а

Используйте этот макрос взамен стандартного malloc'а (парный ему макрос для free легко написать и самостоятельно). Если по выходу из программы gMemCounter не равен нулю, значит, где-то есть утечка. Обратное утверждение в общем случае неверно. Не освобожденная память может сочетаться с двойным вызовом free, в результате чего gMemCounter будет равен нулю, но ведь проблема-то от этого никуда не исчезает! "Лишний" цикл do/while предназначен для обхода конструкций типа: if(xxx) MALLOC(p, s); else yyy; Обойтись без него можно — но тогда фигурные скобки придется везде расставлять явно.

>>> врезка обработка ошибки выделения памяти

Постоянная проверка успешности выделения памяти во-первых, слишком утомительна, во-вторых, загромождает исходный текст, и, в-третьих, приводит к неоправданному увеличению объема откомпилированного кода программы.

```
char *p;
p = malloc(BLOCK_SIZE);
if (!p)
{
        fprintf(stderr,"-ERR: недостаточно памяти для продолжения операци\n");
        _exit();
}
```

Листинг 2 пример неудачной проверки успешности выделения памяти

Одно из возможных решений проблемы сводится к созданию "обертки", вокруг интенсивно используемых функций, проверяющих успешность их завершения и при необходимости рапортующих об ошибке с завершением программы или передающих управление соответствующему обработчику данной аварийной ситуации.

```
void* my_malloc(int x)
{
        int *z;
        z=malloc(x);
        if (!z) GlobalError_and_Save_all_Unsaved_Data;
}
```

Листинг 3 улучшенный пример проверки успешности выделения памяти

>>> врезка устранение фрагментация кучи (динамической памяти)

Можно пойти и дальше, заставив MyMalloc возвращать указатель на указатель. Это позволит дефрагментировать динамическую память, конечно при условии, что адресация блоков всегда будет идти через базовый указатель, который будет постоянно перечитываться из памяти (чтобы компилятор не скэшировал его в регистре, указатель должен быть объявлен как volatile). Как вариант — можно защитить программу критическими секциями, чтобы блок памяти не был перемещен во время работы с ним. И то, и другое снижает производительность, а потому выглядит отнюдь не бесспорным решением.

>>> плоха проверка хуже, чем совсем никакой

Код вида `p = malloc(x) ; if (!p) return 0;` зачастую даже хуже, чем отсутствие какой бы то ни было проверки, т. к. при обращении к нулевому указателю (соответствующему ошибке выделения памяти) операционная система смачно выругается, сообщив причину и адрес проблемы, а кривая проверка просто тихо кончит программу, заставляя окружающих ломать голову — что же такое с ней.

>>> заначка или память прозапас

Виртуальная память системы иногда неожиданно кончается и попытка выделения нового блока посредством malloc'a дает ошибку. В этой ситуации очень важно корректно сохранить все несохраненные данные и выйти. А как быть если для сохранения требуется кусочек памяти? Да очень просто! при старте программы выделяем malloc'ом столько памяти, сколько ее может потребоваться для аварийного сохранения, используя этот НЗ в случае крайней необходимости.

заключение

Количество примочек к GCC (и другим компилятором) растет с каждым днем. Часть из них умирает, часть — растворяется в GCC. Что еще вчера было отдельным проектом, завтра может быть интегрировано с GCC. Поэтому, прежде чем искать нужную вам [тебе] утилиту в сети, имеет смысл поинтересоваться — нет ли чего-то похожего в компиляторе?

Другой хороший источник примочек — дистрибьютивы. В частности, в порты к BSD входит и dmalloc, и даже boehm-gc — автоматический сборщик мусора для Си/Си++ от компании Hewlett-Packard. Тормозит как асфальтовый каток, но зато работает!

Главное, чтобы работа программиста не превращалась в охоту за новыми примочками, ведь в конечном счете, программировать приходится руками и головой.