Принципы Бернштейна для написания безопасного кода

Минко В.С., ИнфоТеКС, отдел ПиР ПАК.

> Я часто вижу, как люди говорят «Никто ещё не создавал неуязвимых программ, поэтому, невозможно написать неуязвимую программу». Следуя этому ложному рассуждению, никто никогда не полетит на Марс, никто не найдёт коллизии хеша MD5, никто не вылечит рак, и т.д.
> — Daniel J. Bernstein

Введение

Дэниел Бернштейн — один из ведущих современных специалистов в мире по компьютерной безопасности. Он известен прежде всего как автор набирающей популярность эллиптической кривой Curve25519 и программ qmail (почтовый сервер) и djbdns (DNS-сервер). Обе программы выполняют стандартный для своего назначения набор функций, могут работать под высокими нагрузками (например, qmail использовался в Yahoo) и приобрели широкую популярность. Однако, главный акцент при написании программ был сделан на безопасность. Вскоре после релиза qmail, Бернштейн даже предложил награду $500 первому, кто опубликует поддающуюся проверке уязвимость в последней версии программы (позже аналогичное заявление было сделано и для djbdns). Через 10 лет после релиза, ни одной уязвимости так и не было найдено [4] (при этом общее число найденных багов было всего 4), после чего Бернштейн увеличил награду до $1000. Стоит также отметить, что обе программы написаны на языке C, который сейчас считается потенциально опасным по сравнению с более высокоуровневыми языками. Такой уровень надёжности и безопасности ПО можно с уверенностью считать беспрецедентным. Поэтому некоторые [1] даже считают Бернштейна лучшим программистом всех времён. Так или иначе, опыт Бернштейна заслуживает внимания и анализа. Настоящая статья далее по тексту объясняет основные принципы, которыми руководствовался Бернштейн при написании своих программ, и благодаря которым он смог достигнуть описанных выше результатов.

Как избежать уязвимости?

«Баг» — это ошибка в программе, приводящая к нарушению пользовательских ожиданий поведения программы. *Уязвимость* — это ошибка в программе, приводящая к нарушению пользовательских ожиданий поведения программы в отношении безопасности. Таким образом, каждая уязвимость — это «баг».

Допустим, что в программе в среднем встречается 1 «баг» на каждые N строк исходного кода. Есть в такой программе суммарно 1000N строк, то можно заключить, что она содержит примерно 1000 «багов», многие из которых — (предположительно) уязвимости.

Каким образом можно добиться, чтобы в программе *не было* уязвимостей? Бернштейн предлагает три ответа на этот вопрос, которые рассмотрены далее по тексту.

Избежание «багов». Первый и наиболее очевидный ответ — это снижение концентрации «багов» в программе. Устранение «багов» — это классическая тема в разработке программного обеспечения. Например, хорошо известно, что радикального снижения числа ошибок в программе можно добиться внедрением юнит-тестов и проведения статического анализа кода. Другим известным способом борьбы с «багами» является проведение тщательного пересмотра исходного кода (code review). Как писал известный программист Эрик Раймонд в своём эссе «Собор и Базар», «при достаточном количестве глаз, все баги всплывают на поверхность». Однако, менее широко известным представляется зависимость числа «багов» от шагов на ранней стадии разработки ПО.

Снижение объёма исходного кода. Допустим, применяя различные методики избежания «багов», их концентрация была успешно снижена с исходной 1 «баг» на N строк кода до 1 «баг» на 10N строк. Однако, этого явно не достаточно, чтоб в 1000N строках кода не содержалось ни одной уязвимости. Второй возможный приём — это снижение объёма кода, реализующего требуемый для пользователя функционал.

Снижение объёма доверенного кода. Допустим, снизив концентрацию ошибок и суммарный объём исходного кода, удалось добиться программы, содержащей менее 100 «багов», предположительно, многие из которых — это уязвимости. Как от них избавиться? Третий предлагаемый метод — это снижение объёма *доверенного кода*. Можно спроектировать такую архитектуру ПО, чтобы большая часть кода помещалась в «недоверенные зоны». Код в «недоверенной зоне» не может нарушить безопасность как бы плохо он ни функционировал, как бы много ошибок он ни содержал.

Таким образом, снизив концентрацию «багов» и общий объём исходного кода, можно полагать, что программа не содержит ошибок в доверенном коде, и, следовательно, является безопасной. Вероятно, в такой программе всё ещё будут содержаться ошибки в недоверенном коде, но они не смогут нарушить безопасность. Поэтому разработка программы, не содержащей «багов» является более сложной задачей, чем разработка безопасной программы.

Далее по тексту каждый их предложенных методов рассмотрен подробнее, а также приведён ряд распространённых заблуждений в отношении достижения поставленной задачи разработки безопасного ПО.

Избежание «багов»

Основным объяснением исключительной безопасности qmail Бернштейн считает малое число ошибок. Этот раздел описывает несколько методик для снижения числа ошибок, использованных при написании qmail и предложенных автором уже постфактум.

Использование ясных потоков данных. Стандартный аргумент против глобальных переменных — это создание ими скрытых потоков данных, часто неожиданных для программиста. Архитектура *qmail* спроектирована таким образом, чтобы сделать внутренние потоки данных максимально ясными. Значительная часть *qmail* выполняются в отдельных UNIX-процессах, соединённых через конвейеры (pipeline), файловую систему и другие механизмы межпроцессного взаимодействия. Но они не имеют прямого доступа к переменным друг друга. Поскольку каждый процесс имеет относительно малое число возможных состояний, существенно снижается вероятность, что программист испортит потоки данных.

Безопасные типы данных. Кому-то покажется странным, но после выполнения `y=x+1`, `y` может быть значительно меньше `x`. Обычно, значения переменных будут соответствовать математическим значениям операций. Однако, в редких случаях — нет. Это произойдёт, если `x` — максимальное для типа данных значение. Для отлова таких случаев

требуется проводить дополнительные вычисления — проверки на превышение границ допустимых значений. Другим вариантом является использование специальных целочисленных библиотек, которые проваливают операции только в случае исчерпания доступной памяти.

Известным источником ошибок в C-программах являются строки, ограниченные нулевым символом. Для исключения таких «багов» Бернштейн использует собственный формат строк, имеющий следующую структуру:

```
typedef struct stralloc
{
    char *s;
    unsigned int len;
    unsigned int a;
}
```

Где `s` — указатель на выделенный буфер для строки, `len` — длина строки, `a` — размер выделенного буфера. Все операции со строками проводятся с использованием этого формата.

Избежание синтаксического анализа. «Парсинг» или синтаксический анализ — это процесс преобразования линейной последовательности лексем в структурированные данные. Зачастую программы для проведения синтаксического анализа (парсеры) содержат ошибки, особенно это характерно для анализа языков со сложной грамматикой. В *qmail* парсинг по возможности избегается. Для межпроцессного взаимодействия и внутренних файловых форматов используется примитивнейший формат, в котором строки в формате *text0* начинаются с одного символа-кода команды [2] (формат text0 означает разделение строк нулевым символом вместо символа переноса строки). В случае, если парсинг полностью избежать нельзя, для отлова ошибок Бернштейн рекомендует каждое синтаксическое правило покрывать отдельным автоматическим тестом.

Обобщение ввода данных. В *qmail* версии 0.90 была исправлена ошибка, вызванная тем, что некорректно обрабатывался результат системного вызова *stat()*. Он может завершиться с ошибкой, если файл существует на файловой системе, но она временно недоступна (например, если используется сетевая файловая система). Автоматические тесты *qmail* не покрывали такие редкие случаи, но проверяли корректность работы в случае возникновения разных ошибок другого рода. Каким образом можно избегать редко проявляющихся ошибок? Тестирование можно существенно упростить, если механизм ввода данных был бы абстрагирован и мог бы взаимодействовать не только с файловой системой. Создание исчерпывающего набора тестов для такого обобщённого механизма ввода было бы простой задачей и легко бы позволило отловить редко-проявляемые ошибки.

Снижение объёма исходного кода

> Если мы захотим считать количество строк кода, мы должны расценивать их не как созданные строки, а как потраченные.
> — Edsger Dijkstra

Выделение общих функций. Рассмотрим следующий отрывок из кода *Sendmail* (почтовый сервер, для которого qmail разрабатывался как безопасная альтернатива):

```
if (dup2(fdv[1], 1) < 0)
{
    syserr("%s: cannot dup2 for stdout", argv[0]);
    _exit(EX_OSERR);
}
close(fdv[1]);
```

Функция *dup2()* делает дубликат дескриптора файла. Конструкция *dup2()/close()* переносит дескриптор из одного места в другое. В Sendmail встречается ряд других мест, где

встречается та же самая конструкция.

В *qmail* эта конструкция вынесена в отдельную функцию *fd_move()*, которая вызывается из десятка других мест в коде:

```
int fd_move(int to,int from)
{
    if (to == from) return 0;
    if (fd_copy(to,from) == -1) return -1;
    close(from);
    return 0;
}
```

Большинство разработчиков не стали бы создавать такую маленькую функцию. Но в итоге, несколько слов кода экономятся на каждом вызове, что суммарно экономит несколько десятков слов кода. Аналогичный подход применим и для функций большего размера.

Автоматическая обработка ошибок. Рассмотрим следующий фрагмент из *qmail/qmail-pw2u.c*:

```
void die_write()
{
    substdio_putsflush(subfderr,"qmail-pw2u: fatal: unable to
write output\n");
    _exit(111);
}
/* ... */
if (substdio_puts(subfdout,uugh) == -1) die_write();
if (substdio_puts(subfdout,dashcolon) == -1) die_write();
if (substdio_put(subfdout,x,i) == -1) die_write();
/* ... */
```

В этом фрагменте используется вывод в стандартный поток ввода-вывода (stdio). Такая операция, как и многие другие, может завершиться с ошибкой. Поэтому в qmail результат каждого без исключения вызова проверяется явным образом. Это приводит к таким негативным последствиям как снижение ясности кода и увеличение его объёма. К сожалению, язык C, на котором написан qmail, не предоставляет других способов обработки ошибок. Однако эта проблема легко решается использованием языков с механизмами обработки исключений. Уже после выпуска qmail Бернштейн признал [3] отказ от C++ (в котором, как известно, механизм обработки исключений присутствует) в пользу языка C неудачным решением.

Использование сетевых утилит. В UNIX имеется специальная утилита для прослушивания сетевых соединений - *inetd*. Когда соединение создано, inet запускает другую программу для его обработки. Так, в случае с qmail запускается *qmail-smtpd* для обработки входящих SMTP-соединений. Поэтому qmail-smtpd не занимается непосредственно сетевым взаимодействием, а лишь считывает SMTP-команды со стандартного ввода и записывает ответы в стандартный вывод.

Sendmail, как и многие другие сетевые программы, сама реализует сетевое взаимодействие. В результате код, предназначенный для выполнения одинаковых по назначению функций, повторяется и, как результат, в каждой отдельной программе получается менее отлаженным. В некоторых случаях требуемый функционал для сетевого взаимодействия немного отличается от возможностей inetd. Например, Sendmail при обработке входящих соединений учитывает загруженность ЦП. Однако и в этом случае Бернштейн считает, что правильным решением была бы доработка inetd, а не повторение реализации раз за разом.

Использование файловой системы. Получив сообщение на адрес username@domain, сервер электронной почты должен найти инструкции по доставке почты для *username*. Обычно для хранения инструкций используется база данных. Sendmail хранит инструкции для всех пользователей в общем файле «*aliases file*». Для извлечения инструкций конкретного

пользователя требуется проведение синтаксического анализа, что, как было отмечено раньше, Бернштейн рекомендует избегать. Поэтому qmail хранит инструкции для пользователя username в файле `.qmail-username`. Таким образом, поиск инструкций для пользователя не требует проведения «парсинга» или других сложных операций, а сводится к простому открытию файла. Аналогичным образом можно снижать объём исходного кода программы и во множестве других случаев, используя файловую систему как готовое хранилище данных в формате ключ-значение.

Использование контроля доступа ОС. Серверы электронной почты в UNIX обрабатывают специальные файлы `.forward`, в которых задаются правила пересылки почты конкретного пользователя. Такие файлы в том числе могут содержать инструкции по вызову сторонних программ. При обработке такого файла Sendmail самостоятельно производит контроль доступа: проверяет, имеет ли пользователь права на доступ к файлу, является ли файл символической ссылкой, запоминает пользователя, от имени которого нужно выполнять указанные в файле пересылки программы и т. п. Реализация ручного контроля доступа получилась сложной и содержала много ошибок. В qmail та же задача решена использованием контроля доступа операционной системы, который уже реализован и хорошо отлажен. Для обработки файла пересылки qmail запускает процесс `qmail-local` с правами того пользователя, чья почта пересылается. `qmail-local` просто считывает файл `.forward`, и при необходимости запускает указанные там программы. Таким образом, ценой вызова одного процесса, удалось добиться значительного упрощения логики обработки файла пересылки и сократить объём кода.

Применение описанных выше методов позволило добиться существенно меньшего объёма исходного кода *qmail* по сравнению с аналогичными MTA.

Таблица 1 — Сравнение объёмов исходных кодов разных почтовых серверов.

MTA	Строк	Слов	Символов	Файлов
qmail-1.01	16 028	44 331	370123	288
sendmail-8.8.8	52 830	179 608	1 218 116	53
zmailer-2.2e10	57 595	205 524	1 423 624	227
smail-3.2	62 331	246 140	1 701 112	151
exim-1.90	67 778	272 084	2 092 351	127

Снижение объёма доверенного кода

Компартментализация и её недостатки. Архитектура qmail была разработана согласно принципу *компартментализации*. Этот принцип призван решить следующую проблему: нарушение безопасности некоторой части системы может привести к эксплуатации других частей системы. Так в Sendmail, которая представляет собой монолитный процесс, ошибка в любой функции может привести к порче любого участка памяти всего процесса. Для решения этой проблемы, программа разделяется на несколько частей, и каждая часть функционирует в своём выделенном домене безопасности [4]. При компрометации одной части системы, другие остаются безопасными. Архитектура qmail (см. рис. 1) представляет собой 9 взаимодействующих процессов, некоторые из которых создаются по запросу, а другие — работают в фоновом режиме («демоны»).

Несмотря на то что такая архитектура — это шаг вперёд по сравнению с монолитной, Бернштейн позднее критиковал её в пользу принципа *снижения объёма доверенного кода* [3]. Так как даже после разделения программы, присутствие «бага», например, в `qmail-remote` может привести к утечке или порче исходящей электронной почты.

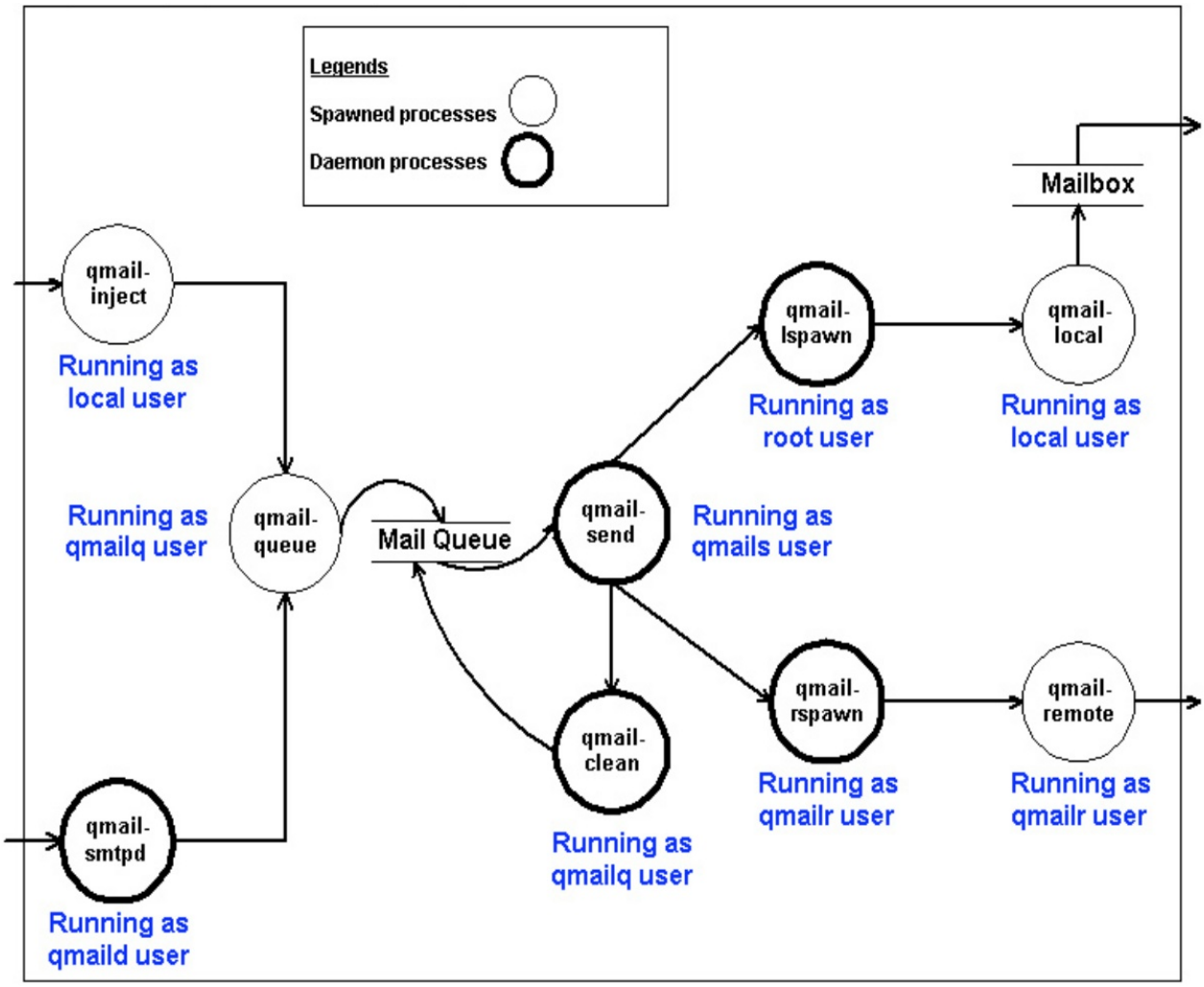

Рисунок 1 — Архитектура qmail.

Вынесение кода в «недоверенные зоны». Принцип снижения объёма доверенного кода проще всего пояснить на примере программы, трансформирующей данные от одного источника. Такой программой может быть, например, просмотрщик изображений или других мультимедийных файлов. Схематично классическая архитектура такой программы представлена на рис. 2.

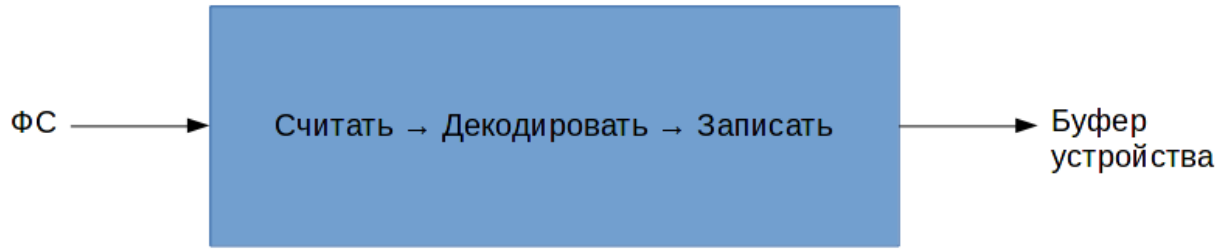

Рисунок 2 — Архитектура программы для просмотра мультимедийных файлов.

Разработчики такого рода программ зачастую не заботятся о безопасности. Однако пользователи могут получать входные файлы для этих программ из недостоверных источников. Если в декодировщике (вероятно, наиболее сложная часть программы) обнаружится ошибка, её эксплуатация может привести к модификации или порче файлов пользователя, т. к. все части программы имеют доступ к файловой системе.

Для исключения таких уязвимостей, предлагается изолировать код декодирования

путём вынесения его в «недоверенную зону». Для этого декодировщик выносится в отдельную программу, которая не имеет доступа к файловой системе, а также ограничена в других системных ресурсах, доступ к которым может привести к нарушению безопасности пользователя (см. рис. 3). Декодировщик считывает закодированный файл со стандартного ввода и записывает обработанные данные (например, растровое изображение) в стандартный вывод. Запись обработанных данных в буфер устройства будет производиться специализированной программой с минимальным объёмом кода, которая хорошо отлажена и имеет доступ к устройству. Таким образом, как бы плохо ни функционировал декодировщик, как бы много ошибок он ни содержал, программа не сможет нарушить безопасность.

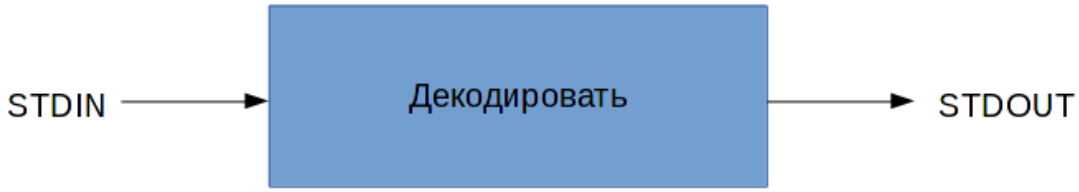

Рисунок 3 — Архитектура программы с изоляцией декодировщика.

Создание «недоверенной зоны» в UNIX. Изоляция программы в UNIX реализуется достаточно просто, для этого нужно выполнить перечисленные ниже операции.

- Запретить создание новых файлов, сокетов и т. п. установкой текущего и максимального значения `RLIMIT_NOFILE`.
- Запретить доступ к файловой системе вызовами `chdir` и `chroot` на пустую директорию.
- Выбрать специальные UID и GID для процесса.
- Убедиться, что ни один другой процесс не выполняется под тем же UID: создать дочерний процесс вызовом `fork`, из которого выполнить `setuid(targetuid)`, `kill(-1, SIGKILL)`, `_exit(0)` и убедиться, что дочерний процесс завершился успешно.
- Запретить `kill()` и `prace()` установкой выбранных UID и GID.
- Запретить создание дочерних процессов установкой текущего и максимального значения `RLIMIT_NOPROC` в 0.
- Установить желаемое ограничение по памяти и другим ресурсам.

После выполнения указанный операций, если в ядре операционной систем отсутствуют ошибки, изолированная программа не будет иметь никаких других каналов взаимодействия кроме исходных файловых дескрипторов.

Распространённые заблуждения

В этом разделе приведён ряд распространённых убеждений, которые Бернштейн считает ложными [3]. Стоит отметить, что в своей критике Бернштейн не затрагивает экономические аспекты вопроса безопасности, а рассматривает всё с позиции теоретика, который ищет подходы для разработки программ, не содержащих уязвимостей.

Препятствие эксплуатации уязвимостей. Для многих безопасность заключается не в том, чтобы избавиться от уязвимостей в коде, а в том, чтобы воспрепятствовать эксплуатации этих уязвимостей. К таковым технологиям можно отнести системы обнаружения вторжения (IDS), методы запутывания (обфускации) кода. Широкую популярность набрали утилиты Enhanced Mitigation Experience Toolkit (EMET) от Microsoft. Утилиты предлагают целый спектр различных методов, направленных на усложнение эксплуатации уязвимостей, основанных на известных принципах (например, DEP — запрет выполнения кода, не

отмеченного в памяти как выполнимый, ASLR — «перемешивание» страниц памяти, чтобы вредоносному коду было сложнее вычислить требуемый адрес, SEHOP — проверка порчи обработчиков исключений и др.). Все эти методы направлены не на исправление ошибок в коде программ, а принимают «баги» как данность и создают ложное впечатление защищённости. Однако при должном усердии все перечисленные методики можно обойти. Например, в 2014-м году исследователи из Bronium Labs продемонстрировали успешный обход всех методов защиты EMET 4.1 [5].

Поэтому Бернштейн заключает, что если мы определяем успех как сдерживание вчерашних атак, вместо того, чтобы прикладывать усилия к предотвращению всех возможных атак, то не стоит удивляться, что наши программы остаются уязвимыми к атакам завтрашнего дня.

Принцип минимизации привилегий. Широко известный принцип минимизации привилегий заключается в том, что каждая программа и пользователь должны функционировать, используя минимальный набор привилегий, необходимый для выполнения своих задач. Этот принцип может снизить ущерб от уязвимости, но почти никогда не устраняет саму уязвимость.

Рассмотрим для примера программу *DNS Helper*, которая предотвращала доступ браузера Netscape к диску. Однако, несмотря на это, обнаруженный «баг» в библиотеке *libresolv* [6] позволил перехватывать контроль над DNS Helper, модифицировать DNS-трафик и перехватывать соединения пользователя. Таким образом, хотя доступ к диску и был изолирован, наличие уязвимости всё равно привело к нарушению безопасности пользователя.

Преобразование DNS Helper в «недоверенную зону» — более комплексная мера, чем наложение ограничений на ресурсы ОС, доступ к которым имеет программа. Если программа обрабатывает данные от нескольких источников, каждый источник должен быть изолирован от модификации данных других источников.

Пренебрежение безопасностью ради скорости. Нередко разработчики отвергают изложенные выше методы Бернштейна как априори неприемлемо замедляющие программу:

- Запуск отдельного процесса для использование контроля доступа операционной системой или изоляции синтаксического анализа;
- Использование файловой системы как хранилища вместо СУБД.

Бернштейн считает безопасность гораздо более важным качеством, чем быстродействие: сначала следует обеспечить отсутствие уязвимостей и уже потом беспокоиться о производительности. Многим это утверждение покажется спорным. Так, разработчики высоконагруженных систем рекомендуют ещё при разработке архитектуры ПО учитывать вопросы быстродействия. Тем не менее, когда был выпущен первый релиз qmail, в которой были использованы описанные методы Бернштейна, производительность почтового сервера оказалась примерно вдвое выше, чем у Sendmail актуальной версии. Поэтому, возможно, степень негативного влияния на производительность от методов Бернштейна на первый взгляд представляется завышенной.

Список использованных источников

1. Swartz A., «*djb (Aaron Swartz's Raw Thought)*» - http://www.aaronsw.com/weblog/djb, 2009.
2. Bernstein D., «*The qmail security guarantee*» - http://cr.yp.to/qmail/guarantee.html, 2007.
3. Bernstein D., «*Some thoughts on security after ten years of qmail 1.0*», 2007.
4. Hafiz M., Johnson R., Afandi R., «*The Security Architecture of qmail*», 2004.
5. DeMott J., «*Bypassing EMET 4.1*», 2014.
6. Computer emergency response team, «*CA-2002-19: Buffer Overflows in Multiple DNS Resolver Libraries*», 2002.