MAGAZINE

BSD

FOR NOVICE AND ADVANCED USERS

BSD SECURITY

UPGRADE YOUR SKILLS

INSIDE

ANATOMY OF FREEBSD COMPROMISE – HOW TO DETER AND TRACK ATTACKERS
DNS SECURITY – DIFFERENT THREATS TO DNS TRANSACTIONS
QJAIL & ZFS – HOW TO SECURE THE HOST SYSTEM
SYNCHRONIZATION PROBLEMS – HOW TO USE THE SLEEP MUTEX
POSTGRESQL – SERVER-SIDE PROGRAMMING PART 2
ZFS MADNESS WITH BEADM – HOW TO

VOL.5 NO.06
ISSUE 06/2012(35)
1898-9144

NEW SERIES

iXsystems' New Server Line Featuring the Intel® Xeon® Processor E5-2600 Family:

The Future of Enterprise-Grade Servers

MODEL: iXR-22x4IB

http://www.iXsystems.com/e5

KEY FEATURES

iXR-1204+10G

- Dual Intel® Xeon® E5-2600 Family Processors
- Intel® C600 series chipset
- Intel® X540 Dual-Port 10 Gigabit Ethernet Controllers
- Up to 16 Cores and 32 process threads
- Up to 768GB main memory
- Four SAS/SATA drive bays
- Onboard SATA RAID 0, 1, 5, and 10
- 700W high-efficiency redundant power supply with FC and PMBus (80%+ Gold Certified)

iXR-22x4IB

- Dual Intel® Xeon® E5-2600 Family Processors per node
- Intel® C600 series chipset
- Four server nodes in 2U of rack space
- Up to 256GB main memory per server node
- One Mellanox® ConnectX QDR 40Gbp/s Infiniband w/QSFP Connector per node
- 12 SAS/SATA drive bays, 3 per node
- Hardware RAID via LSI2108 controller
- Shared 1620W redundant high-efficiency Platinum level (91%+) power supplies

Call iXsystems toll free or visit our website today! **1-855-GREP-4-IX | www.iXsystems.com**

iXsystems is pleased to present a range of new, **blazingly fast servers** designed to meet all the demands of today's increasingly data-intensive corporate world.

HIGH Throughput & INCREDIBLE Performance Density

Based on the Intel® Xeon® Processor E5-2600 Family and the Intel® C600 series chipset, the new server line represents a revolutionary upgrade in performance and throughput.

With a combination of technologies including Intel® Integrated I/O, Intel® Data Direct I/O Technology, and Intel® Turbo Boost Technology, the innovative microarchitecture of the Intel® Xeon® Processor E5-2600 Family delivers up to 80% greater performance over previous-generation processors and increases energy efficiency, making the servers well-suited for the most demanding applications. In addition to integrated support for PCIe 3.0, the new chipset also features integrated SAS to provide low-latency, high throughput access to data.

MODEL: IXR-1204+10G – 10GbE On-Board

The new iXR-1204+10G features Dual Intel® Xeon® E5-2600 Family Processors, up to 768GB of RAM, and dual onboard 10GigE NICs in a single rack unit, utilizing space effectively while supplying more processing power than ever before. The high performance density of the iXR-1204 +10G makes it suitable for clustering, high-traffic webservers, virtualization, and cloud computing applications.

For computation and throughput-intensive applications, iXsystems now offers the iXR-22x4IB. Boasting four server nodes in 2U of rack space, each node features dual Intel® Xeon® E5-2600 series processors, up to 256GB of RAM, and one Mellanox® ConnectX QDR 40Gbp/s Infiniband with a QSFP Connector. The iXR-22x4IB is perfect for high-powered computing, virtualization, or business intelligence applications that require the computing power of the Intel® Xeon® Processor E5-2600 Family and the high throughput of Infiniband.

MODEL: IXR-22x4IB

Call iXsystems toll free or visit our website today! **1-855-GREP-4-IX | www.iXsystems.com**

Dear Readers,

As well as previous issue(s), June issue, focuses on BSD security. I hope that there are not too many articles dedicated to this topic out there. If you are interested in this field, you will find many interesting articles that would help you upgrade your skills. As we've mentioned in the last issue, the security field is really interesting part of our life – desired, useful and practical field to explore. By reading articles related to this area, you not only improve your admin skills, but also secure your system and data. We know that this area is one of the most wanted and in June issue, you can discover several good pieces of work made by our authors. I hope you will enjoy reading them.

If you are keen on DNS security, just go to page 6 and check out the next part of Paul Ammann's article.

In Part 2, Paul looks at protocol-based threats to the operation and administration of DNS. Also, he'll examine DNS protocol methods to address these threats. Non-DNS protocol based solutions such as IPSec are beyond the scope of this article, but could be a more appropriate solution for an organization based on its infrastructure.

Also, I would like to present the two series published by BSD magazine.

On page 12, you'll find the article written by Rob Somerville. This time Rob shows you how to examine different defensive methods to deter and track attackers. In part 6 you will also read how to examine some techniques that can assist in identifying and delaying attacks.

Second one is Luca Ferrari's article about PostgreSQL: Server-Side Programming. In the previous article readers have learnt how to write simple triggers and stored procedures using plpgsql PostgreSQL extension to SQL. In June, Perl will be used as a language for both triggers and procedures showing how PostgreSQL can be flexible for server-side programming.

Next comes Joseph Kong with his Synchronization Problems or: How I Learned to Stop Worrying and Love the Sleep Mutex. Joseph addresses his article to the problem of data and state corruption caused by concurrent threads.

And for all of those who want more, just go to page 44 to see the article written by Edward Tan on Upgrading Ports Using Portmaster. You will learn about portmaster and the ports tree, find out more on the infinity loop of upgrading ports as well as you find some tips and tricks of using portmaster.

So, again this issue is more security related. I hope it is something that you expected.

If you still need more, we have special offer for you. Just go to page 11 and see the books list published by No Starch Press. They've prepared the special code for BSD readers.

Now, you can buy Joseph Kong's book at lower price, so if you like his article and would like to learn more, just use the code NEWBUS to get 30% discount.

If you like such offers, just let me know – if you like our articles or would like to share some of your expectations – write me back.

Enjoy Reading!
Ewa Dudzic
& BSD Team

MAGAZINE BSD

Editor in Chief:
Ewa Dudzic
ewa.dudzic@software.com.pl

Contributing:
Kris Moore, Carlos Antonio Neira, Benedikt Niessen, Jesse Smith, Giovanni Bechis, Luca Ferrari, Rob Somerville, Michael Shirk, Paul Ammann

Top Betatesters & Proofreaders:
Paul McMath, Bjorn Michelsen, Barry Grumbine, Eric De La Cruz, Luca Ferrari, Imad Soltani, Norman Golisz, Sander Reiche, Mahesh J., Rob Cabrera, Pablo Halamaj, Cleiton Alves

Special Thanks:
Denise Ebery

Art Director:
Ireneusz Pogroszewski

DTP:
Ireneusz Pogroszewski
ireneusz.pogroszewski@software.com.pl

Senior Consultant/Publisher:
Paweł Marciniak *pawel@software.com.pl*

CEO:
Ewa Dudzic
ewa.dudzic@software.com.pl

Production Director:
Andrzej Kuca
andrzej.kuca@software.com.pl

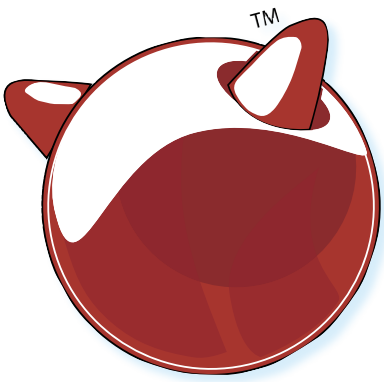
Executive Ad Consultant:
Ewa Dudzic
ewa.dudzic@software.com.pl

Advertising Sales:
Patrycja Przybyłowicz
patrycja.przybylowicz@software.com.pl

Publisher :
Software Press Sp. z o.o. SK
ul. Bokserska 1, 02-682 Warszawa
Poland
worldwide publishing
tel: 1 917 338 36 31
www.bsdmag.org

Software Press Sp z o.o. SK is looking for partners from all over the world. If you are interested in cooperation with us, please contact us via e-mail: *editors@bsdmag.org*

Security



How To

Developer's Corner

ZFS

DNSSEC: Threats to DNS Transactions Part 2

The threats to a DNS transaction depend on the type of transaction. Name resolution queries and responses (DNS query/response) between DNS clients (stub resolver or resolving name server) and DNS servers (caching/resolving name server or authoritative name server) could involve any nodes in the Internet.

What you will learn…

- Different threats to DNS transactions and how to mitigate them

What you should know…

- You should have some knowledge and background of DNS and how it works

Hence, the threats against them are much greater in number and severity compared to those for zone transfer, dynamic update, and DNS NOTIFY transactions. In general, the nodes involved in zone transfer, dynamic update, and DNS NOTIFY transactions are all within the administrative domain of a single organization. The [only] exceptions are instances in which the primary or secondary name servers of an organization are run on its behalf by ISPs or other organizations. There usually is a preexisting trust relationship in these cases, however, so it is not difficult to set up a mutual authentication system for DNS zone transfers.

In this article, we're going to look at protocol-based threats to the operation and administration of DNS. Also, we'll examine DNS protocol methods to address these threats. Non-DNS protocol based solutions such as IPSec are beyond the scope of this article, but could be a more appropriate solution for an organization based on its infrastructure.

DNS Query/Response

DNS name resolution queries and responses (DNS query/response) generally involve single, unsigned, and unencrypted UDP packets. The known threats to DNS query/response transactions have been documented in IETF RFC 3833 and can be classified as follows:

- *Threat #12*: Forged or bogus response
- *Threat #13*: Removal of some RRs from the response
- *Threat #14*: Incorrect expansion rules applied to wildcard RRs in a zone file.

Forged or Bogus Response

A forged or bogus response is a response which is different from what the legitimate authoritative name server expects. A bogus response can originate from:

- A compromised authoritative name server (for queries originating from a resolving name server)
- A poisoned cache of a resolving name server (for queries originating from a stub resolver).

An authoritative name server could be compromised by a platform-level attack on its OS or communication stack (see the May 2012 issue).

The cache of a resolving (caching) name server could be poisoned by the following attacks:

- *Packet Interception.* In this type of attack, the attacker eavesdrops on a request and is able to generate and send a response by spoofing an authoritative name server before the real response from the legitimate authoritative name server reaches the resolving name server.

- *ID Guessing and Query Prediction*. In this type of attack, the attacker guesses the ID field in the header of the DNS request message (because this field is only 16 bits long, brute force guessing is possible) and possibly the QNAME and QTYPE (owner name and RRType, respectively). The attacker then injects bogus data into the network as a response by spoofing a name server.
- *Responses Accumulated from a Compromised Authoritative Name Server*. A compromised authoritative name server is directed by a controlling adversary to send out bogus responses to queries from resolving name servers.

The impacts on a system serviced by a resolving name server that has a poisoned cache are as follows:

- *Denial of Service*. If some crucial RRs such as address records (A RRs) are forged, the system that requires this information can never establish connectivity with the intended node.
- *Client Redirection through Cache Poisoning*. Client redirection is performed by selective poisoning of DNS RRs whose RDATA element contains a name. Examples of such RRs are CNAME, NS, and MX. The name resolution (i.e., IP address) information for these names is found in a set of additional information (or *glue records* when discussing a delegation response). Normally the resolving name server obtains these necessary A/AAAA RRs through follow-up queries (also called *triggered queries*). The responses flowing into the network from these follow-up queries present yet another opportunity for the attacker to insert bogus records. First the attacker can introduce arbitrary names of the attacker's choosing in the RDATA portion of selected RRs; then the attacker can insert the IP addresses of servers (chosen by the attacker) in associated glue records that are transmitted as an answer to follow-up queries. This type of attack on two sets of related responses is called a *name chaining attack*. The overall effect of poisoning the cache of a resolving name server this way is to misdirect several clients who are making use of the services of that resolving name server. Redirecting the users to nodes of the attacker's choosing may enable the attacker to capture sensitive information such as passwords.

Removal of Some RRs

Apart from injecting bogus or forged data in a response, an attacker could also remove RRs from a response. This action might result in a name resolution query failure and consequent denial of service.

Incorrect Expansion Rules Applied to Wildcard RRs

Many zones use wildcard RRs to economize on the volume of data in the zone file. The wildcard patterns are used for synthesizing RRs on the fly in generating responses for name resolution queries. (The synthesis rules are outlined in section 4.3.2 of IETF RFC 1034.) If synthesis rules are applied incorrectly in a name server, the RRs associated with resources existing in an organization may not be generated and made available in a DNS response. This fault also results in denial of service.

Protection Approach for DNS Query/Response Threats

The underlying feature in the major threat associated with DNS query/response (i.e., forged response or response failure) is the integrity of DNS data returned in the response. Hence, the security objective is to verify the integrity of each response received. An integral part of integrity verification is to ensure that valid data has originated from the right source. Establishing trust in the source is called *data origin authentication*. Hence, the security objectives – and consequently the security services – that are required for securing the DNS query/response transaction are data origin authentication and data integrity verification.

These services could be provided by establishing trust in the source and verifying the signature of the data sent by that source. The specification for a digital signature mechanism in the context of the DNS infrastructure is in IETF's DNSSEC standard. The objectives, additional RRs, and DNS message contents involved in the DNSSEC are specified through RFCs 4033, 4034, and 4035. In DNSSEC, trust in the public key (for signature verification) of the source is established not by going to a third party or a chain of third parties (as in public key infrastructure [PKI] chaining), but by starting from a trusted name server (such as the root name server) and establishing the chain of trust down to the current source of response through successive verifications of signature of the public key of a child by its parent. The public key of the trusted name servers is called the *trust anchor*.

After authenticating the source, the next process DNSSEC calls for is to authenticate the response. This requires that responses consist of not only the requested RRs but also an authenticator associated

with them. In DNSSEC, this authenticator is the digital signature of an RRSet. The digital signature of an RRSet is encapsulated through a special RRType called RRSIG. The DNS client using the trusted public key of the source (whose trust has just been established) then verifies the digital signature to detect if the response is valid or bogus.

To ensure that RRs associated with a query are really missing in the zone file and have not been removed in transit, the DNSSEC mechanism provides a means for authenticating the nonexistence of an RR. It generates a special RR called an NSEC RR that lists the RRTypes associated with an owner name as well as the next name in the zone file. It sends this special RR, along with its signature, to the resolving name server. By verifying this signature, a DNSSEC-aware resolving name server can determine which authoritative owner name exists in a zone and which authoritative RRTypes exist at those owner names.

To protect against the threat of incorrect application of expansion rules for wildcard RRs, the DNSSEC mechanism provides a means of comparing the validated wildcard RR against an NSEC RR and thereby verifying that the name server applied the wildcard expansion rules correctly in generating an answer.

DNSSEC can guarantee the integrity of name resolution responses to DNS clients acting on behalf of Internet-based resources, provided the clients perform the DNSSEC signature verification. In many cases, however, these DNS clients are stub resolvers that are not DNSSEC-aware. If signature verification is performed by the resolving name server providing name resolution service for the clients that are stub resolvers, the end-to-end integrity of the response data can be guaranteed only by protecting the communication channel between the resolving name server and the stub resolver.

IETF's design criteria consider DNS data to be public; hence, confidentiality is not one of the security goals of DNSSEC. DNSSEC is not designed to directly protect against denial-of-service threats, although it does so indirectly by providing message integrity and source authentication. DNSSEC also does not provide communication channel security because name resolution queries and responses travel over millions of nodes of the public Internet. DNSSEC also can lead to a new type of weakness that did not exist in DNS before. An artifact of how DNSSEC performs negative responses allows a client to map all the names in a zone. This is called Zone Walking. Zone Walking provides an attacker with a "map" of a target zone with all domain names and IP addresses in the zone and enables him/her to determine the configuration of the internal network and launch some targeted attacks on some key hosts. Therefore, it is advisable that a zone only contains zone data that the administrator wants to be made public. For internal DNS, something like split-DNS could be deployed.

Zone Transfer

Zone transfers are performed to replicate zone files in multiple servers to provide a degree of fault tolerance in the DNS service provided by an organization. Threats from zone transfers have not been documented formally through any IETF RFCs. A few threats could be expected, however: the first threat, denial of service, is common for any network transaction. The second threat is based on exploitation of knowledge gained from the information provided by zone transfers. The third threat is common to any network packet.

- *Threat #15 – Denial of Service*: Because zone transfers involve the transfer of entire zones, they place substantial demands on network resources relative to normal DNS queries. Errant or malicious frequent zone transfer requests on the name servers of the enterprise can overload the master zone server and result in denial of service to legitimate users.
- *Threat #16*: The zone transfer response message could be tampered.

The denial-of-service can be minimized if servers allowed to make zone transfer requests are restricted to a set of known entities. To configure this restriction into the primary name server, there should be a means of identifying those entities. Name server software such as BIND initially provided a configuration feature to restrict zone transfer requests to a set of designated IP addresses. Because IP addresses can be spoofed, however, this mode of configuration does not provide an adequate means of restricting zone transfer access.

The IETF developed an alternate mechanism called a transaction signature (TSIG), whereby mutual identification of servers is based on a shared secret key. Because the number of servers involved in zone transfer is limited (generally restricted to name servers in the same administrative domain of an organization), a bilateral trust model that is based on a shared secret key may be adequate for most enterprise (except for very large ones). TSIG specifies that the shared secret key

be used not only for mutual authentication but also for signing zone transfer requests and responses. Hence, it provides protection against tampering of zone transfer response messages (threat T15). Protection of DNS data alone (the payload) in a zone transfer message also can be ensured through verification of signature records accompanying RRs from a DNSSEC-signed zone. These signatures, however, do not cover all the information in a zone file (e.g., delegation information). Furthermore, they enable verification of only the individual RRsets and not the entire zone transfer response message.

There is also another method to authenticate DNS transactions by using asymmetric cryptography (i.e. public key cryptography). The format of the SIG(0) RR is similar to the resource record signature (RRSIG) RR, and can be validated using a public key stored in the DNS (instead of a shared secret key). SIG(0) can be more computational expensive to use, but offer an advantage in that a previous trust relationship may not be necessary to use SIG(0) signed messages. However, since most zone transfers occur between parties that have a previously established relationship, it is considered easier to implement TSIG for authenticating zone transfer transactions.

Alternatives to TSIG

Although TSIG is widely deployed, there are several problems with the protocol that you should be aware of:

- It requires distributing secret keys to each host which must make updates.
- The HMAC-MD5 digest is only 128 bits.
- There are no levels of authority. Any host with the secret key may update any record.

As a result, a number of alternatives and extensions have been proposed:

- RFC 2137 specifies an update method using a public key "SIG" DNS record. A client holding the corresponding private key can sign the update request. This method matches the DNSSEC method for secure queries. However, this method is deprecated by RFC 3007.
- In 2003, RFC 3645 proposed extending TSIG to allow the *Generic Security Service* (GSS) method of secure key exchange, eliminating the need for manually distributing keys to all TSIG clients. The method for distributing public keys as a DNS *resource record* (RR) is specified in RFC 2930, with GSS as one mode of this method. A modified GSS-TSIG, using the Windows Kerberos Server, was implemented by Microsoft Windows Active Directory servers and clients called Secure Dynamic Update. In combination with poorly configured DNS (with no Reverse Lookup Zone) using RFC 1918 addressing, reverse DNS updates using this authentication scheme are forwarded en masse to the root DNS servers and increase the traffic to root DNS servers in the course of doing so[1]. There is an anycast group which deals with this traffic to take it away from the root DNS servers [2].
- RFC 2845, which defines TSIG, specifies only one allowed hashing function HMAC-MD5, which is no longer considered to be highly secure. As of 2006, proposals are being circulated to allow RFC 3174 Secure Hash Algorithm (SHA1) hashing to replace MD5. The 160-bit digest generated by SHA1 should be more secure than the 128-bit digest generated by MD5.
- RFC 2930, which defines TKEY, a DNS Record used to automatically distribute keys from a DNS server to DNS clients.
- RFC 3645, which defines GSS-TSIG which uses gss-api and TKEY to automatically distribute keys in gss-api mode.
- The DNSCurve proposal has many similarities to TSIG.

Dynamic Updates

Dynamic updates involve DNS clients making changes to zone data in an authoritative name server in real time. Clients typically performing dynamic updates are CA servers, DHCP servers, or Internet Multicast Address servers. As with zone transfer transaction, the threats associated with dynamic update transaction have not been officially documented by the IETF through an RFC. The following are some common threats that could be expected, based on the fact that dynamic updates involve a data update request transiting a network.

- *Threat #17 – Unauthorized Updates*: Unauthorized updates could have several harmful consequences for the content of zone data. Some harmful data operations include: (a) adding illegitimate resources (new FQDN and new RRs to a valid zone file), (b) deleting legitimate resources (entire FQDN or specific RRs), and (c) altering delegation information (NS RRs pointing to child zones).

- *Threat #18*: The data in a dynamic update request could be tampered.
- *Threat #19 – Replay Attacks*: Update request messages could be captured and resubmitted later, thus causing inappropriate updates.

Threats #16 and #17 could be countered by authenticating the entities involved and providing a means to detect tampering of the messages. Because these security objectives in the case of zone transfer are met by the TSIG/SIG(0) mechanism, the same TSIG/SIG(0) mechanism is specified for protecting dynamic updates.

Although the dynamic update message contains some replay attack (Threat #18) protection in the prerequisite field of the message, TSIG/SIG(0) provides an additional mechanism to protect against replay attacks by including a timestamp field in the dynamic update request.

This signed timestamp enables a server to determine whether the timing of the dynamic update request is within the acceptable time limits specified in the configuration.

It sometimes makes more sense to use SIG(0) protection mechanisms for dynamic update than for zone transfer. Dynamic update transactions may happen between parties that do not always have a prior security relationship or may be part of a bootstrapping operation. Therefore it may be impractical to use TSIG with a shared secret, but SIG(0) authentication using keys stored in the DNS may be a possibility.

Another possibility is to rely on lower level network layer to provide security such as IPSec. This would remove the need for authentication at the DNS (application) layer. How to set up this level of security is beyond the scope of this guide.

DNS Notify

DNS NOTIFY is a message sent by primary (master) name servers to secondary (slave) name servers, causing the secondary servers to start a refresh operation (e.g. query for SOA RR to check the serial number, etc.) and perform a zone transfer if an update to the zone has occurred. Because the NOTIFY message is only a signal, there are only minor security risks in dealing with the message. The primary security risk to consider is the following:

- *Threat #20 – Spurious NOTIFY Messages*: Secondary name servers would receive spurious DNS NOTIFY messages from sources other than the primary name server.

Footnotes

1. *http://www.caida.org/publications/papers/2003/dnsspectroscopy/*
2. *http://public.as112.net/*

The only impact of receiving spurious DNS NOTIFY message is the increase in workload in secondary name servers since a zone transfer will only occur when an updated zone is on the primary server. Because this threat is low impact, the only protection approach required is to configure the secondary name servers to receive DNS NOTIFY message only from the enterprise's primary name server. However, if TSIG is set up for use for all communication between a set of hosts, TSIG will be used with NOTIFY messages as well.

Summary

Let's recap what we've learned in this article.

There are a number of threats to DNS query/response: forged or bogus responses, removal of records (RRs) in responses, and incorrect application of wildcard expansion rules. Your security objectives should be data origin authentication and data integrity verification.

With zone transfers, you should be concerned with denial of service and tampering of messages. Your security objective is mutual authentication and data integrity verification.

Dynamic update is susceptible to unauthorized updates, tampering of messages, and replay attacks. Focus should be on mutual authentication, data integrity verification, and signed timestamps.

In Part 3, we look at how to secure the DNS hosting environment.

PAUL AMMANN

Paul Ammann lives in New Fairfeld, CT with his wife and 4 cats. You can reach him at pq_aq (at) fastmail (dot) us.

Anatomy

of a FreeBSD Compromise Part 6

In the final part in our series, we will examine different defensive methods to deter and track attackers.

What you will learn…
- Common techniques used to deter hackers

What you should know…
- BSD and network administration skills

While it is impossible to secure a server against every possible form of attack that the dark side may muster, by taking defensive steps the system administrator can make life exceedingly difficult for the hacker and can delay if not totally avoid a successful attack. While many of the suggestions in this article are probably second nature to most admins, it cannot be stressed enough with busy schedules and tight deadlines the importance of preventative maintenance which has a tendency to slip down the priority list. We will also examine some techniques that can assist in identifying and delaying attacks.

Steps to practical security

Is your software up to date?

Old software with documented vulnerabilities is the hackers dream. Even if steps are taken to prevent the identification of services, with the automated tools available to the attacker, taking the gamble that a well documented exploit may work is trivial. While patching is not the full solution (and software regression is always a possibility), this step closes the door to the well documented exploits. It is also always worth testing new software and patches in a development environment to see what weaknesses are present. The author recently installed a major CMS and was writing a validation routine for a database query, and as part of the testing of the code discovered a bug in the core software. The test (typing garbage into an HTML test field) was handled correctly by the new module, but the problem lay further back up the chain. While this "manual" discovery revealed a weakness, a software fuzzer would be a more efficient way of testing. The fact remains that an un-patched system is more of at risk than an patched one.

Reduce your available footprint

Apart from the bloat, is that service/software/application essential? Running services 24/7 that are only occasionally required increases risk. 5 minutes to write a shell script that puts a server into maintenance mode and runs remote services such as Webmin and SSH for remote access is one quick workaround. Alternatively, run SSH only and turn services on as required remotely. It is a good plan to "daisy chain" access, e.g. only have SSH running on one machine and tunnel through to other machines once inside the network (with a separate user login/password for an additional layer of security).

A good firewall / proxy is essential

An additional layer of security is vital on any public facing Internet connection. Even at the most rudimentary level, traffic shaping, packet inspection, malformed request

rejection etc. can mitigate some crude attacks, but it is not the complete solution. From the administration point of view, a single dedicated server that centrally deals with security is good practice, and this can be expanded with Intrusion Detection Systems (e.g. Snort), honey-pots and tar pits as required. How sophisticated this appliance is depends very much on the architecture and services available on your network, but not all firewall technology is equal. See (Table 1) What is deep inspection?

Move the door handle

Unless you are running a service which must run on a standard port (e.g. HTTP or SMTP), consider relocating the service to a high non-standard port. Most casual scans do not bother past port 1024. The author manages a number of web servers, and one web application runs on a high port rather than port 80. To date, the amount of suspicious traffic in the Apache logs is zero. Other Apache boxes get at least 5 probes per day on the same domain. Also consider the market of your ISP / comms provider, mass market "consumer" providers with lots of un-patched PC's will have a lot more noise than a provider that is geared towards the business sector.

Be observant

Logs, stats and email alerts are an essential aid in identifying incidents. Develop an intuition for the extraordinary – once you are familiar with the usage patterns on your network, anything out of the ordinary will quickly become apparent. Make sure your logs are backed up and taken off-line at regular intervals, as this may provide useful evidence if an attacker does gain access and manages to obscure the trail by deleting their activity. Alternatively, use a dedicated log server or flag critical files with the append only flag *sappend*.

Tune your environment

Don't rely on the default settings provided by software out of the box to provide security. A good example is SSH, Version 1.0 is better than Telnet, but limiting your server to V 2.0 connections is a better approach. Limit access to "those that really must". Consider limiting access on time basis (e.g. disable internal proxies out of business hours). Chrooting critical applications such as webservers etc. on multiple service servers is a good idea. Use encryption where possible, and when developing sensitive web applications help your visitors by automatically redirecting to a secure domain and disallowing unencrypted traffic.

Keep a baseline

System backups, binary file checksums using utilities such as Tripwire, or even a totally read-only system environment are essential on critical systems. Run regular scans such as chkrootkit or rkhunter to identify rootkits or suspicious binary files. Gently portscan your network from another network on a regular basis to identify any open ports that may have appeared due to changed configurations. Keep a log of changes e.g. patch history, modifications and software updates.

Security by obscurity

Many flame-wars can be found on the Internet on the subject of security by obscurity. As an approach alone, it is seriously flawed as the determined hacker will not be distracted by minor modifications to a system that do not address core security. However, it does provide the system administrator with an additional layer that may delay the more popular and common attacks. A "quiet" server which does not announce services on common ports and obscures software versions will be harder to analyze, and false trails can be laid using honey-pots etc. The other side of the coin is that an observant hacker will pick up on the fact that the system administrator has taken the trouble to disguise what is really going on and will consider the server a valuable target.

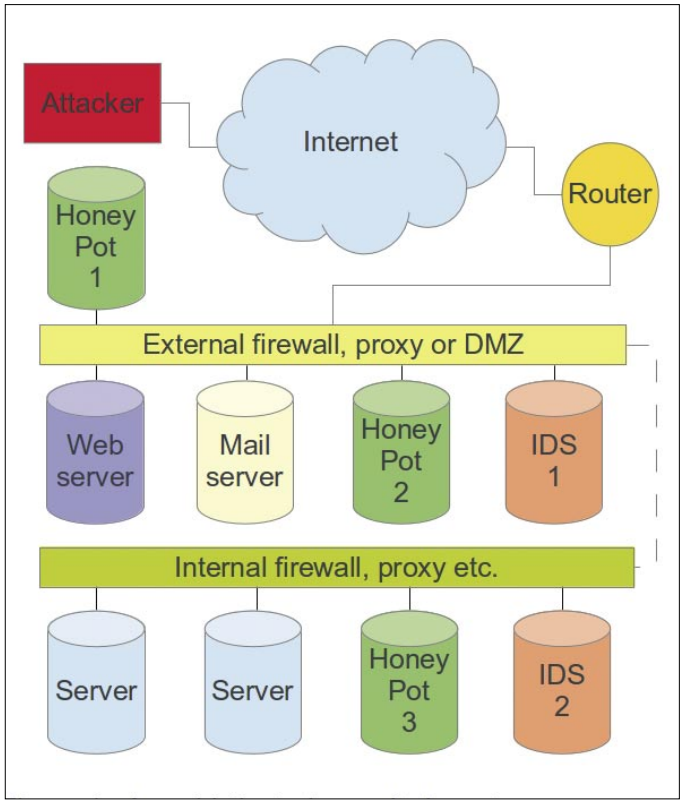


Figure 1. *A sophisticated security layout*

Used alone, security by obscurity is not a viable method to protect a server. Used with other techniques, it can buy the administrator some time and facilitate additional information gathering on the attacker.

Honeypots, tar pits and Intrusion Detection Systems

A honeypot is a dedicated sever that acts as a "sacrificial target" and a deliberate lure for attackers. A number of different strategies can be employed, depending on exactly what the administrator wants to achieve. The basic premise is that the honeypot exposes various dummy services (e.g. web-server, database, email etc) to the attacker so that information can be gathered about the strategies used to compromise networks and collect information on the attackers IP address, etc. A honeypot can be purely sacrificial in that sufficient authority is given to the attacker so that they can destroy the box, or with some creative scripting and firewall rules the honeypot can be more proactive e.g. return a portscan the attacker. A tar pit is a variation on the honypot theme, in that it is "sticky" – e.g. a telnet service is exposed to the attacker, and instead of timing out, the connection is persistent, so the attack is effectively trapped and contained – like an insect on flypaper. The IDS on the other hand monitors suspect traffic on the wire, depending on a set of rules. The most common solutions include Honeyd, Labrea and Nessus or Snort. A complex layout is illustrated in Figure 1.

Honeypot 1 is totally sacrificial, lying outside the firewall it is totally exposed to any attack and is used to establish a baseline and a resource for researching attacks on this particular network. Honeypot 2 is used to gather information on traffic that has passed through to the DMZ, and Honeypot 3 any serious attacks that reach the internal network. The 2 IDS monitor illicit traffic at the DMZ and inside the network. With firewall rules, it is possible to re-direct any undesired requests to the honeypot, which will further confuse and delay the attacker. For example, where a port scan is initiated on port 25 (SMTP) of the webserver, the firewall could redirect port 25 traffic to the honeypot. As desired, the administrator can replace honeypots with tar-pits, depending on how they wish to handle hacker activity.

While this layout is excessive for a small network, it demonstrates the different roles of each device. It is up to the administrator to decide the best strategy, depending on the resources available. Running a honeypot requires commitment, and is not a solution in itself. Depending on your locality, the legal position may be a very gray area, as technically they honeypot or tar-pit is intercepting the attackers communications. There is also the matter of what to do with the information gathered – is there a process for dealing with this (Fire back at the attacker or just monitor?) and when is an attack considered serious enough to engage law enforcement? The answer to these will depend very much on the network and the organization.

Mod Security

Mod security is a web application firewall that runs under Apache and can provide a further layer of security to a webserver. Providing both a positive and a negative rule model (e.g. only valid requests are accepted or known bad requests are rejected) Mod Security can be tailored exactly to the security model the administrator wishes to enforce. With some tuning, it can even bounce the rejected request to a honeypot, tar-pit or another external site.

Lets get practical

The following setups were performed on a FreeBSD 9.0-RELEASE box running Apache 2.2.21 and OpenSSH 5.8 running on port 22. Due to the restrictive nature of `Mod_security`, test this out on a development box first or run it in DetectionOnly mode as the restrictive rule set will probably break your webserver.

Installing Mod_Security

```
cd /usr/ports/www/mod_security21
make install clean
cd /usr/local/etc/apache22/Includes/mod_security2
cp modsecurity_crs_10_config.conf config-crs-10.000
mkdir /var/log/modsecurity
chown www:www /var/log/modsecurity
```

Edit the `modsecurity_crs_10_config.conf` file to show:

```
SecRuleEngine On
SecDebugLog /var/log/modsecurity/modsec_debug.log
SecAuditLog /var/log/modsecurity/modsec_audit.log
```

Amend `http.cond` to read:

```
LoadFile /usr/local/lib/libxml2.so
LoadModule security2_module libexec/apache22/mod_security2.so
```

Restart Apache:

```
/usr/local/etc/rc.d/apache22 restart
```

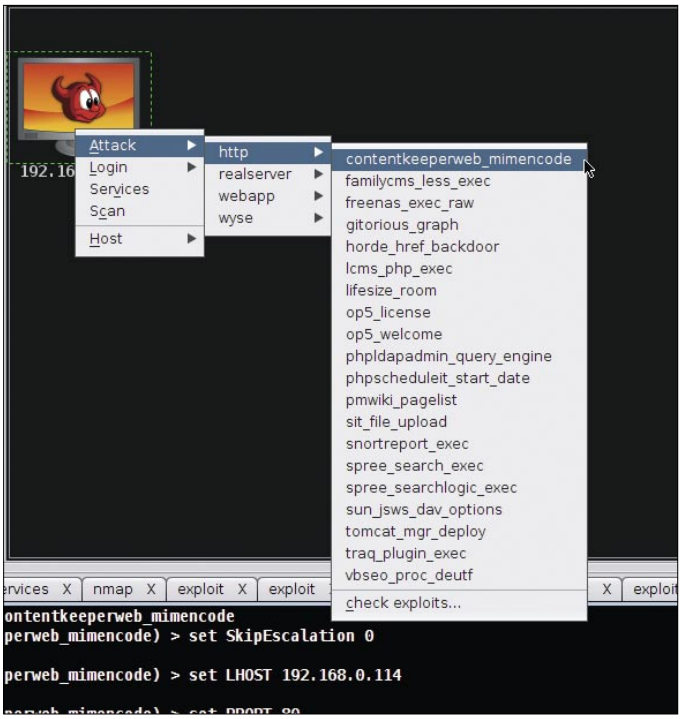


Figure 2. *Attacking the host with Armitage*

Check that your webserver still works by pointing a browser at port 80 on the target machine. You may need to comment out the following line in `modsecurity_crs_21_protocol_anomalies.conf` with a `#` if you are accessing the test machine via an IP address rather than a host name:

```
SecRule REQUEST_HEADERS:Host „^[\d\.]+$”
„deny,log,auditlog,status:400,msg:’Host header is a
numeric IP address’,
severity:’2’,id:’960017’”
```

Spin up Backtrak Armatage and fire an attack at the test box (Figure 2). If you view `modsec_audit.log` you should see the following similar entries (Figure 3):

- Rogue request is sent from 192.168.0.114
- `Mod_security` closes the connection with a 500 error
- The pattern match why the request was rejected
- The action carried out

The Tar Pit

Labrea will populate your network with dummy tar-pitted hosts using ARP. Use with caution on a live network. As the developer says labrea can break things in the network. So you are encouraged to read the README. For those that are impatient though:

Install and run Labrea (replace `em0` with your network interface as appropriate):

```
cd /usr/ports/security/labrea
make install clean
LaBrea -z -s -o -b -p 10000 -i em0
```

Using nmap, your host should report whatever ports open you have running software on. Now pick an IP address that you you know is not allocated on your network (In my case 192.168.0.132) and ping it: Listing 1.

Hmm. What happens if we run NMAP against it? See (Figure 4). According to NMAP, I have telnet running (sic) what happens when I try and access it?

```
telnet 192.168.0.132
Trying 192.168.0.132...
Connected to 192.168.0.132.
Escape character is ‘^]’.
```

Figure 3. *Mod security log shows 500 error returned*

Listing 1. *LaBrea in action*

```
ping 192.168.0.132
PING 192.168.0.132 (192.168.0.132) 56(84) bytes of data.
From 192.168.0.103 icmp_seq=1 Destination Host Unreachable
From 192.168.0.103 icmp_seq=2 Destination Host Unreachable
From 192.168.0.103 icmp_seq=3 Destination Host Unreachable
64 bytes from 192.168.0.132: icmp_req=4 ttl=255 time=196 ms
64 bytes from 192.168.0.132: icmp_req=5 ttl=255 time=95.6 ms
64 bytes from 192.168.0.132: icmp_req=6 ttl=255 time=94.3 ms
64 bytes from 192.168.0.132: icmp_req=7 ttl=255 time=33.2 ms
```

Listing 2. *Installing Kippo*

```
su
pkg_add -r py27-twisted py27-pycrypto
wget http://kippo.googlecode.com/files/kippo-0.5.tar.gz
exit
tar -xvzf kippo-0.5.tar.gz
cd kippo-0.5
ls
```

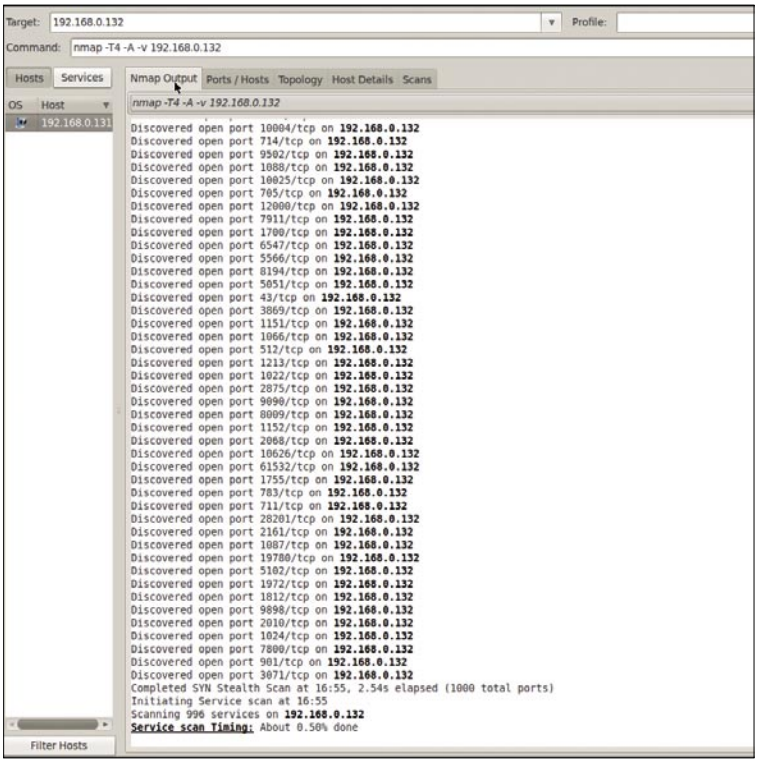


Figure 4. *NMAP found 1000 ports but is struggling to get O/S signature etc.*

Labrea has created a tar-pit telnet session, complete with successful connection string but no connection.

A SSH honeypot

Secure Shell is the standard for remote CLI access to servers. Encrypted end-to-end traffic, the ability to use password or key authentication makes this essential software in the Administrators toolkit. Unfortunately, the popularity of SSH means it is vulnerable to probing by attackers, and if insecure passwords are used (or root access not disabled) it attracts the wrong sort of attention.

One useful strategy is to install Kippo, a SSH honeypot based on Python that will not only act as a dummy SSH server, but will also log all the attackers session activity which will help the administrator identify what strategy the attacker is adopting and what malware or rootkits they are installing. If required, Kippo can be run on a high port (e.g. 2222) and SSH requests on port 22 redirected using IPFW of PF etc. Kippo emulates the Linux filesystem, but this can ne changed if desired. Execute the following: Listing 2.

The following directories and files will be extracted:

- `dl/` – files downloaded with wget are stored here
- `log/kippo.log` – log/debug output
- `log/tty/` – session logs
- `utils/playlog.py` – utility to replay session logs
- `utils/createfs.py` – used to create fs.pickle
- `fs.pickle` – fake filesystem
- `honeyfs/` – file contents for the fake filesystem

Now run kippo on port 2222 of your honeypot:

```
./start.sh
```

Kippo will generate a RSA key pair. Spin up Nmap or Zenmap and scan your server, you should see port 2222 open (Figure 5). Let us now “compromise” our honeypot and install and run a linux rootkit (Replace 192.168.0.131 with the destination IP address of your honeypot): Listing 3.

Kippo rejects the request, and all the “attackers” commands are logged (Figure 6). If you are feeling particularly malevolent, you can execute:

WARNING – Ensure you are in sales: before you execute this destructive command!

```
rm -fr /*
```

Listing 3. *Kippo honeypot session*

```
ssh root@192.168.0.131 -p2222
The authenticity of host '[192.168.0.131]:2222
                  ([192.168.0.131]:2222)' can't be
                  established.
RSA key fingerprint is ea:75:13:cc:8e:98:c9:e3:e3:72:74:
                  93:99:5e:01:fb.
Are you sure you want to continue connecting (yes/no)?
                  yes
Warning: Permanently added '[192.168.0.131]:2222' (RSA)
                  to the list of known        hosts.
Password: 123456

sales:~# cd /
sales:/# ls -alh

drwxr-xr-x 1 root root     0 2009-11-20 08:19 sys
drwxr-xr-x 1 root root  4096 2009-11-08 15:42 bin
drwxr-xr-x 1 root root  4096 2009-11-06 11:08 mnt
drwxr-xr-x 1 root root  4096 2009-11-06 11:08 media
                      [...]
-rwxrwxrwx 1 root root    25 2009-11-06 11:16 vmlinuz

sales:/# wget www.tigerlne.netfast.org/rk.tgz

--2012-05-07 20:06:20--  http://
                  www.tigerlne.netfast.org/rk.tgz
Connecting to www.tigerlne.netfast.org:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 754474 (736K) [application/x-gzip]
Saving to: 'rk.tgz

100%[=====================================>] 754,474
                  132K/s  eta 0s

2012-05-07 20:06:25 (132 KB/s) - 'rk.tgz' saved
                  [754474/754474]

sales:/# tar -xvzf rk.tgz

rk-asdqw
[...]
rk-asdqw/bin

sales:/# cd rk-asdqw /bin
sales:/rk-asdqw# ls -alh

drwxr-xr-x 1 root root   4096 2007-07-16 15:48 bin
drwxr-xr-x 1 root root   4096 2007-07-16 15:48 conf
                                [...]
-rw-r--r-- 1 root root   3200 2004-11-26 14:18 setup

sales:/rk-asdqw# ./setup sillypassword 10077
 ___
{o,o}
|)__)
-"-"-
O RLY?
```

Listing 4. *SSH guard in action*

```
ssh 192.168.0.131
Password: yourpasswordhere
Last login: Mon May  7 19:26:18 2012 from 192.168.0.103
FreeBSD 9.0-RELEASE (GENERIC) #0: Tue Jan  3 07:15:25
                  UTC 2012
Welcome to FreeBSD!
exit
Connection to 192.168.0.131 closed.

ssh 192.168.0.131
Password:
Password:
Password:
Permission denied (publickey,keyboard-interactive).

ssh 192.168.0.131
Password:
Password:
Password:
Permission denied (publickey,keyboard-interactive).
ssh 192.168.0.131
ssh_exchange_identification: Connection closed by remote
                  host
```

Listing 5. *SSH guard log*

```
May  7 20:00:00 hacker sshguard[1139]: Started successfully [(a,p,s)=(40, 420, 1200)], now ready to scan.
May  7 20:00:42 hacker sshd[1156]: error: PAM: authentication error for testuser from 192.168.0.103

May  7 20:00:51 hacker sshd[1161]: error: PAM: authentication error for testuser from 192.168.0.103
May  7 20:00:51 hacker sshguard[1139]: Blocking 192.168.0.103:4 for >630secs:
40 danger in 3 attacks over 9 seconds (all: 40d in 1 abuses over 9s).
May  7 20:00:52 hacker sshd[1161]: error: PAM: authentication error for testuser from 192.168.0.103
May  7 20:00:52 hacker sshd[1161]: error: PAM: authentication error for testuser from 192.168.0.103
May  7 20:01:04 hacker sshd[1167]: refused connect from 192.168.0.103 (192.168.0.103)
May  7 20:08:17 hacker sshd[1178]: Accepted keyboard-interactive/pam for testuser
from 192.168.0.254 port 62276 ssh2
```

Your current Kippo session will not allow you to exit, but if you re-attach with another SSH session, sales will be undamaged.

Rejecting failed SSH logins

Using the utility SSHguard the administrator can run a normal SSH session and monitor via syslog for any unauthorized activity in `auth.log`. Ensure you have direct physical console access or another machine with a different IP address and SSH installed before running this – when you lock yourself out of SSH you will be locked out of your server.

```
pkg_add -r sshguard
```

Now edit your `rc.conf` file to reflect the following:

```
syslogd_enable="YES"
syslogd_flags="-ss" # local log
```

Edit `syslog.conf` to run SSHguard when unauthorized activity observed:

```
# Added for SSHguard support
auth.info;authpriv.info  |exec /usr/local/sbin/sshguard
```

Reboot:

```
reboot
```

You should be able to login via SSH on the remote host as normal. Exit from your SSH session, and attempt another SSH login but this type press [Enter] at the password prompt: Listing 4. If you now examine `/var/log/auth.log` you should see similar entries to this: Listing 5.

SSH access is now disabled to 192.168.0.103 for 10 minutes. Access from 192.168.0.254 was not restricted. If desired, SSHguard can be configured to use firewall rules to totally deny access to all ports from the attacker – see *http://www.sshguard.net*.

In conclusion

There are many counter-measures the administrator can deploy to confuse, track and investigate hackers. The biggest hurdle in implementing a robust security environment is the time it takes to adequately test and tune the modifications for each individual server or network especially in a production environment.

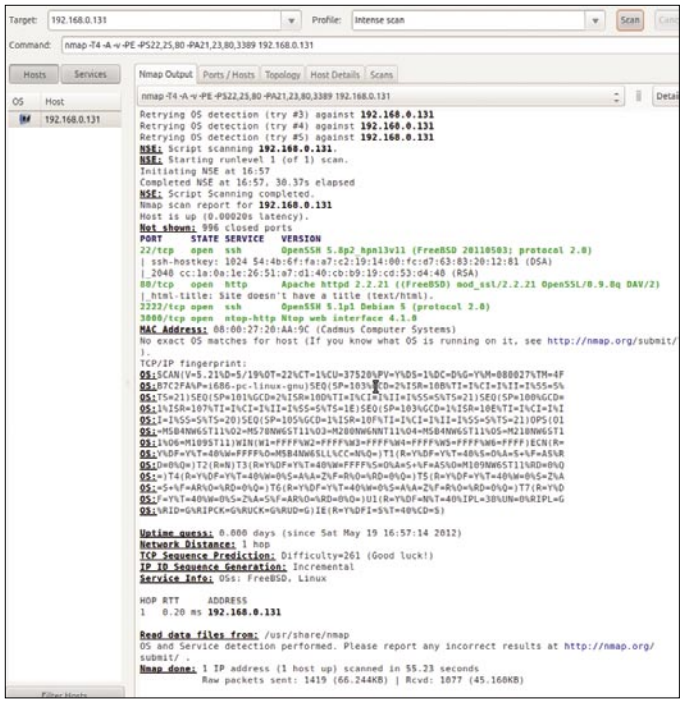


Figure 5. *NMAP found port 2222 open (NTOP running on port 3000)*

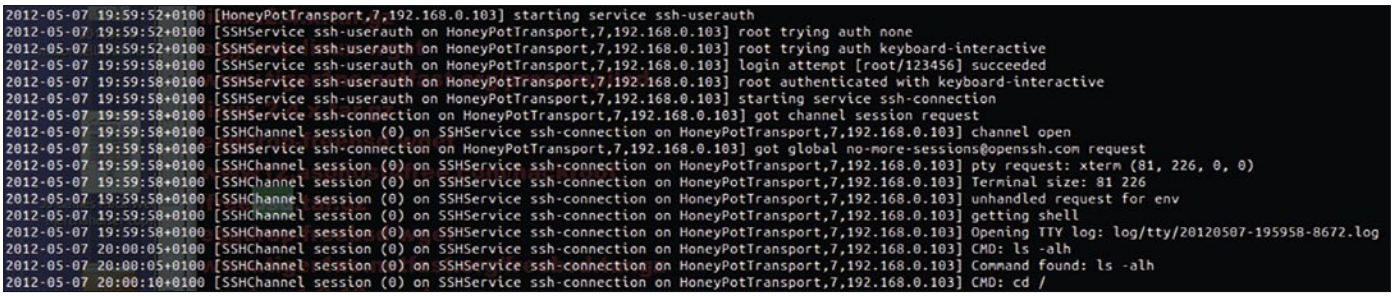


Figure 6. *Kippo logfile*

Table 1. *Further reading and resources*

Further reading	
Description	URL
Security through obscurity Ain't What They Think It Is	*http://web.archive.org/web/20070202151534/http://www.bastille-linux.org/jay/obscurity-revisited.html*
Six dumbest ideas in computer security	*http://www.ranum.com/security/computer_security/editorials/dumb*
What is deep inspection? A discussion about DPI versus proxy firewalls	*http://www.ranum.com/security/computer_security/editorials/deepinspect/index.html*
Honeypots – Tracking the hackers	*http://books.google.co.uk/books?id=xBE73h-zdi4C*
Mastering FreeBSD and OpenBSD security	*http://books.google.co.uk/books?id=gqKwaHmXp4YC*

Table 2. *FreeBSD security software resources*

Security software	
Description	URL
Tripwire – File system security and verification program	*http://sourceforge.net/projects/tripwire*
Sshguard – Protect hosts from brute force attacks against ssh and other services	*http://www.sshguard.net*
Labrea – Security tarpit defense tool	*http://labrea.sourceforge.net/labrea-info.html*
Honeyd – Simulate virtual network hosts (honeypots)	*http://www.honeyd.org*
Kippo – SSH honeypot	*http://code.google.com/p/kippo/*
Modsecurity – An intrusion detection and prevention engine	*http://www.modsecurity.org*
Hardening FreeBSD	*http://www.bsdguides.org/guides/freebsd/security/harden.php*

Ultimately, it is a balancing act between practicality and risk, and it may be more appropriate to roll out a solution that causes the least disruption but catches 85% of the attacks automatically with the remaining 15% being monitored by manual traditional methods of observation. The closer you get to perfection, the greater the chance that inadvertently something will break, or worst still, false positives will be generated.

Using the arsenal of tools documented in this series gives the administrator the ability not only to identify issues, but opens the Pandora's box of the security world – how to remain one step ahead of an enemy who never sleeps and is constantly on the lookout for new opportunities. Unfortunately, IT security is shrouded in a lot of hype, and as many front line support engineers will testify, products sometimes do not perform as expected. Thankfully, the *BSD community is well ahead of the curve with an attitude of "security first". Hopefully, other sectors of the software industry will follow suit, as collectively we cannot rise above the lowest common denominator.

ROB SOMERVILLE

Rob Somerville has been passionate about technology since his early teens. A keen advocate of open systems since the mid eighties, he has worked in many corporate sectors including finance, automotive, airlines, government and media in a variety of roles from technical support, system administrator, developer, systems integrator and IT manager. He has moved on from CP/M and nixie tubes but keeps a soldering iron handy just in case.

Using Qjail to set up the basejail

FreeBSD's jail system offers process isolation within a separate environment in order to secure the host system. In case of a compromised service, only the jail running that service is affected. In a similar fashion, ZFS allows the creation of a separate filesystem for each jail.

What you will learn…

- The basics of setting up jails using qjail
- Managing jails with the tools qjail provides

What you should know…

- The fundamentals of jails and what they were designed to do
- Creating ZFS filesystems and setting specific options for them

This adds to the power of jails as each jail can have its own set of filesystem parameters such as quotas and reservations. While ZFS is relatively easy to set up, jails usually needs a bit more effort. This article explains how jails can be used a third party wrapper script called Qjail. Together with ZFS, it offers great flexibility in handling multiple jails and can even save some storage space.

What is Qjail?

The FreeBSD handbook has a whole chapter about setting up jails. This usually involves a lengthy buildworld and installworld process, which takes a lot of time. To reduce the build time to practically zero, wrappers have been created as third party software. With Qjail, a jail can be set up quickly (hence the preceding q) without requiring a whole buildworld and installworld run beforehand. In addition, it offers management functionalities like starting and stopping jails, listing all jails currently configured on the system and even instantiating multiple jails at once.

Qjail was developed by Joe Barbish and released under the BSD license. While it is basically a set of powerful shell scripts, it is not very difficult to learn or use. To install it on FreeBSD, use the ports collection where it resides under *sysutils/qjail*. It comes with extensive documentation including examples for each subcommand in the form of two man pages (`qjail-intro(8)` and `qjail(8)`) which explain the basic philosophy behind its development and how to make use of it, respectively.

Qjail does not require a full `buildworld/installworld` cycle to create a jail. Instead, it downloads official FreeBSD releases from the mirror servers provided by the project. Since these releases contain everything needed to run a complete FreeBSD system, it can also be used for jails. Qjail simply downloads a FreeBSD release as a basic jail called basejail, from which each new jail is spawned.

Qjail is capable of creating two different kinds of jails: directory based jails and sparse image file based jails. Directory based jails share the same disk space as the host and can theoretically grow to the maximum available disk space of the drive it's installed on. Sparse image file jails are different in that their size must be set at the time of their creation, but only occupy as much space as they currently need. This article explains directory based jails only. However, sparse image file based jails are not difficult to set up and are explained in great detail in the `qjail(8)` man page.

Qjail also makes use of the nullfs filesystem to link files that are common to the basic jail infrastructure from the host system. This reduces the overhead of copying the files into each jail and saves a lot of disk space. It also simplifies jail management. For example, when an updated library is installed, all jails automatically use that library once they are restarted since they are all using a link to it provided by nullfs.

Using Qjail to Set up the basejail

Before we can create a jail, we need to set up our system so that it can hold our basejail. The basejail and all the other jails we create from it will be stored in a separate ZFS filesystem. This article assumes that you've already set up your zpool with one or more disks (substitute `<poolname>` in the examples below with your actual pool name). Qjail uses the directory `/usr/jails` to set up its directory structure. Since we'll be using ZFS, we create this directory structure ourselves by typing:

```
# zfs create -o mountpoint=/usr/jails -o compression=
on <poolname>/usr/jails
```

Note that if your `/usr` directory is already on a ZFS filesystem, you might need to adjust the mountpoints above accordingly. You can also use a different kind of compression algorithm or higher/lower compression levels to fit your needs (depending on what kind of files you store in the jail). This is just to show you that the files placed in there are good candidates for compression. Activating deduplication using `zfs set dedup=on data/usr/jails` is optional, but the more jails you have on the same filesystem, the more you can benefit from this space-saving feature as there are many files that can be deduplicated.

Next, we use the `qjail` command to download and install a minimal basejail, without a source tree and manual pages. This will slim down the size of our jails. Installing the basejail needs to be done only once. For this example, we are going to fetch an official release from the FreeBSD FTP servers. It will be installed into our ZFS filesystem we've just created. The command is `qjail install` and the output is shown in Listing 1.

Qjail notifies us that our current system is not based on an official FreeBSD release and provides a listing of files on the FTP server which it thinks can be used as an install source. We're going to use 8.2-RELEASE as the basis for our jails (we will update them to a more current release later in this article) and enter this at the prompt. Next, a rather lengthy and verbose FTP fetch session will occur, which has been replaced by … to shorten the listing. After the fetch operation has finished, four new directories have been created in `/usr/jails`: *archive*, *basejail*, *newjail*, and *flavors*. The *archive* directory is used when archiving a jail, which we don't cover here. You can find more information about archiving jails in qjail(8). In *basejail*, all the base system files we just downloaded are being stored. It should be very familiar, since it is basically everything a FreeBSD system needs to run and what will later be part of the jail as well (refer to hier(7) if neccessary). You can trim down the jails a bit by deleting files and directories you know you won't need in your jails. For example, it is safe to delete the `/usr/jails/basejail/boot` directory, as jails do not go through a real boot process and therefore don't need a kernel and loadable modules. The *newjail* directory has symlinks into the basejail directory for `bin/`, `boot/` (obsolete as well when

Listing 1. *Fetching and creating the basejail infrastructure*

```
Your system is 9.0-PRERELEASE.

Normally FTP-servers don't provide non-RELEASE-builds.

Querying your ftp-server...

The ftp server you specified (ftp2.freebsd.org) provides the following RELEASE distributions:...Select one.
7.4-RELEASE
8.2-RELEASE
ISO-IMAGES
README.TXT
amd64
Release [ 9.0-PRERELEASE ]: 8.2-RELEASE
…
Basejail & newjail are being populated.
Est LT 1 minute elapse time for this to complete.
Successfully installed qjail system.
```

deleting the *boot* directory from the *basejail*), `lib/`, `libexec/`, `sbin/`, and `sys/`. This will be used to instantiate a new jail. The last of the four directories is called *flavors*, and is helpful if you want to create certain kind of jails that are configured in a similar fashion. For example, if you want to make sure that certain settings are made in rc.conf in each of the jails, you could create a certain flavor for this configuration. When creating a jail, you can tell qjail that this new jail should be based on a certain flavor and all the settings in *rc.conf* will be created automatically based on your settings in the flavor script. In our example, Qjail will use a default flavor with no special jail settings.

Spawning a jail from the basejail

Now that the basic infrastructure of qjail is installed, we can create our first jail called testjail. Each new jail will be created below the `/usr/jails` directory, so our testjail will reside in `/usr/jails/testjail`. To make management a little easier, we'll use ZFS to set an (arbitrary) storage limit for the jail before creating it.

```
# zfs create -o quota=200m -o reservation=200m -o
compression=on <poolname>/usr/jails/testjail
```

Make sure that the ZFS filesystem has exactly the same name that you are going to give to the jail. Depending on how much storage space the jail is going to need, you can raise or lower the quota and reservation being shown here. Next, we create the jail using the `qjail create` command shown in Listing 2.

Qjail created a new jail with the IP alias 192.168.0.2 on `em0` (substitute these values with the NIC and network configuration at your site) and named it testjail. A new directory structure below `/usr/jails/testjail` has been created as a result of this command. If you compare the directories that have been created, you'll discover that the symlinks are the same as in the *newjail* directory, since this is being used a a kind of template for each new jail. This is just one jail, but you can create multiple jails with Qjail using the command in Listing 3.

The -D option followed by a number up to 100 adds a suffix number for the current jail to the jail name, like prison-1, prison-2, and so on. Using the option -I in conjunction with the -D option will increase the IP address octet by one so that each jail gets its own IP alias assigned to em0. In our example, prison-1 will have the IP address 192.168.0.3, prison-2 will get 192.168.0.4 and prison-3 is reachable on 192.168.0.5 once the jails are running. This is very convenient when you need to quickly create multiple jails and don't want to create aliases first. The aliases are created on the specified NIC when the jail is started and will also be removed again once the jail is stopped.

Listing 2. *Creating a jail*

```
# qjail create -n em0 testjail 192.168.0.2
Successfully created  testjail
```

Listing 3. *Creating multiple jails*

```
# qjail create -n em0 -D 3 -I prison 192.168.0.2
Successfully created  prison-1
Successfully created  prison-2
Successfully created  prison-3
```

Listing 4. *Creating multiple jails*

```
# qjail start testjail
Jail started successfully. testjail
# qjail start
Jail already running.      testjail
Jail started successfully. prison-3
Jail started successfully. prison-2
Jail started successfully. prison-1
```

Listing 5. *Showing the status of all running jails*

```
STA JID  NIC IP              Jailname
--- ---- --- --------------- ------------------------
DR  1    em0    192.168.0.2      jailname
DR  2    em0    192.168.0.5      prison-3
DR  3    em0    192.168.0.4      prison-2
DR  4    em0    192.168.0.3      prison-1
```

Starting and Stopping jails

It's simple to start, stop, or restart jails with Qjail. Use the name of the jail in the `qjail start` command to run a single jail or leave it out to start all jails that are not running currently. Listing 4 shows this for our example jails created earlier. Now, we have all the jails running on our system. Note that you need to work with the jailnames, as Qjail does not accept the jail IDs that are listed in the output of `jls`.

Checking the Status of Running jails

Qjail also provides a command to list all running jails and show some associated information like the IP-address currently assigned to each jail. To show this list, use the command `qjail list`. An example output is provided in Listing 5.

The first column named STA displays the status of the jail in that line. In this example, the D indicates that this jail is directory based. If you were using sparse image file based jails, an I would be shown instead. R indicates that this jail is currently running. An S would have told us that this jail has been stopped.

The next field is JID, which shows the ID of the jail. This is the same unique identifier that `jls` will list for that particular jail, but only if it is running. The NIC field shows the network interface that is associated with the jail and which is handling the traffic. If this field is empty, as opposed to say `em0` or `fxp1`, it means that said jail has no NIC configured. Until we change that using the `qjail configure` command detailed below, this jail will not be able to do any networking. The following field, IP, is obviously the IP address alias that this jail is using. Lastly, the Jailname column shows the name that the jail has been assigned at the time of its creation.

Accessing the jail From the Host System

Now that a jail has been created and is running, we can connect to the console to do basic system administration tasks as the root user. To do that, use the following command to get access to a root shell within the jail: `qjail console testjail`. To quit that root jail (log out), simply use the same commands you would use to quit your host system's shell like `exit`, `logout` or ^D (CTRL+D). Even if there are no users available in the system yet, we can use this command to access the jail's console at any time.

Making Ports Available in the jail

Since the jails will host the services we want to isolate form the base system, we need a way to install sofware using the ports collection. As such, we need to have access to our own ports tree within the jail. While we cannot use the ports tree from the base system (since that would allow

Listing 6. *Updating the ports tree in the jail*

```
# qjail update -p testjail

Sun Apr 15 13:36:36 CEST 2012

The elapse download time of the portsnap compressed ports file
is estimated at 25 minutes for the initial fetch.
Subsequent fetches will generally take less than a minute.

Looking up portsnap.FreeBSD.org mirrors... 4 mirrors found.
Fetching snapshot tag from portsnap1.FreeBSD.org... done.
Fetching snapshot metadata... done.
Updating from Sat Apr 14 13:34:31 CEST 2012 to Sun Apr 15 13:25:17 CEST 2012.
Fetching 4 metadata patches... done.
Applying metadata patches... done.
Fetching 0 metadata files... done.
Fetching 102 patches.....10....20....30....40....50....60....70....80....90....100. done.
Applying patches... done.
Fetching 5 new ports or files... done.
Portsnap fetch completed successfully

Sun Apr 15 13:36:48 CEST 2012

The ports basejail/usr/ports directory tree is being updated.
The elapse time for this to complete is estimated at 1 minute
to 10 minutes depending on how current your ports system is.

Portsnap update completed successfully
```

Listing 7. *Updating the base system binaries of the jail*

```
Error: All jails have to be stopped. This jail is
                    running. testjail

# qjail stop testjail
Jail stopped successfully. testjail

# qjail update -b testjail
Deletion of basejail binaries successful for bin.
Deletion of basejail binaries successful for boot.
Deletion of basejail binaries successful for lib.
Deletion of basejail binaries successful for libexec.
…
Deletion of basejail binaries successful for usr/
                    lib32.

Copied host's binaries to basejail successfully for
                    bin.
Copied host's binaries to basejail successfully for
                    boot.
Copied host's binaries to basejail successfully for
                    lib.
…
Copied host's binaries to basejail successfully for
                    usr/lib32.

Host to basejail binaries update completed
                    successfully.

# qjail start testjail
Jail started successfully. testjail
```

Listing 8. *Reconfiguring a jail not to be run at boot time*

```
qjail config -r norun testjail
Successfull set norun testjail
```

Listing 9. *Renaming a jail*

```
qjail config -n webjail testjail
Successfully renamed  testjail
```

Listing 10. *Assigning a different IP address to a jail*

```
qjail config -i 127.0.0.1 prison-1
Successful ip change prison-1
```

jails access outside of the environment they are restricted to), we can easily make it available using qjail. If no ports tree is installed yet, qjail will invoke `portsnap` to fetch the complete ports tree and make it available for the jails in the directory `/usr/jails/basejail/usr/ports`.

Listing 6 shows the update of an already installed ports tree using `portsnap`. After the ports tree has been updated, you can log into the jail as shown above and run your favorite utility to install ports in the jail.

Updating jails managed by qjail

Remember the 8.2-RELEASE we had to fetch in the `qjail install` example above? Suppose we have been using our Qjail-managed jails for quite some time and a new version (major or minor) of FreeBSD has been released. Since we want to make use of the updates and potential security fixes as well as performance and stability improvements in our jails, Qjail can help with updating the jails. First, the host system needs to be updated using either `freebsd-update` or make `buildworld/installworld`. Once that has been completed, we can tell Qjail to replace the jail system binaries with the updated host system binaries. To do that for our testjail, issue the following command: qjail update -b testjail whose output is shown in Listing 7.

First, we will be notified that these binaries cannot be replaced while the jail is still running. Hence we use `qjail stop testjail` to stop it. Repeat the update command again and it will detect that the jail has been stopped and will begin deleting the old binaries in the jail and copying the new files from the host system. For each system folder being updated, a status message will be printed. After the host's binaries have all been copied without any errors, the last thing we need to do is to start the jail again using the command `qjail start testjail`. It will now start with the updated system binaries and we have completed updating this jail.

Configuring the jails

The only thing left to do is making sure that the jails will automatically start when the host system is rebooted. To do that, you need to add the following line to `/etc/rc.conf` in the host system: `qjail_enable="YES"`. After that, each jail will be started when the host is booting. In case you do not want to start all of your jails, you can exclude some of them with the `qjail configure` command. Listing 8 will show this for our testjail.

Another way to use the `qjail configure` command is to assign another name to a jail. Let's say our testjail is running some webserver and we want to reflect that in its name. Since this would affect the underlying ZFS filesystem as well, we need to rename it first:

On the 'Net

- *http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html* – The FreeBSD handbook chapter about jails,
- *http://qjail.sourceforge.net/* – The homepage of Qjail.
- *http://qjail.sourceforge.net/Qjail-Intro.htm* – qjail-intro(8) man page
- *http://qjail.sourceforge.net/Qjail%20Manual.htm* – qjail(8) man page

Glossary

- Host system: The system that will host the jails and from which jails are being managed.
- Base system: These are the utilities that are shipped with the operating system such as cp, ls, and top among many others. They are also available in the jails.
- Jail: Isolates processes using sophisticated enhancements to the chroot(8) facility to run a separate system inside the host system.

`zfs rename <poolname>usr/jails/testjail <poolname>/usr/jails/webjail`. Then we can rename the testjail (after we've stopped the jail first) with the command in Listing 9.

To change the IP address assigned to a jail, you can also use the `qjail config` command as shown in Listing 10.

With that, we have seen most, but not all of the tools that Qjail provides for creating and managing jails effectively and easy. ZFS adds more to the power by compressing and deduplicating the storage space these jails are using, as well as limiting jails from allocating too much space by using quotas and reservations. Make sure to read the qjail man page as there are many more interesting and powerful things you can do with this jail management utility.

BENEDICT REUSCHLING

Benedict Reuschling has been using FreeBSD since 5.2.1-RELEASE. He was lurking quietly on the FreeBSD mailing lists until 2008 when he joined the FreeBSD German Documentation team. After having received his commit bit for the FreeBSD project's documentation set as well, he's been involved in many aspects of documentation around FreeBSD. Having worked in the private sector at a number of companies during his time as a student, he now enjoys teaching students at the Department of Computer Science of his alma mater where he spent so much time with his own education. During his spare time, he is practicing the Tai Chi Yang style to relax and get his mind off of things.

PostgreSQL:

Server-Side Programming Part 2

In the previous article readers have learnt how to write simple triggers and stored procedures using plpgsql PostgreSQL extension to SQL.

What you will learn…

- Server-side programming with PostgreSQL
- How to run Perl code within PostgreSQL
- How to use the listen-notify IPC

What you should know…

- Basic SQL concepts
- Basic PostgreSQL concepts
- Difference among stored procedures, triggers and rules

In this article Perl will be used as a language for both triggers and procedures showing how PostgreSQL can be flexible for server-side programming. Finally, the listen/notify IPC mechanism embedded into the database will be presented.

Using Foreign Languages

One great advantage of PostgreSQL is that it can run functions written in several foreign languages other than pure *sql* and its extension *plpgsql* or the standard C. There are extensions that allow developers to write procedures using Java, Perl, Python and even Bash-like scripting! Chances are that an extension to support your favourite programming language in PostgreSQL has been already written.

PostgreSQL allows a foreign language to operate in two different ways: a “trusted” and an “untrusted” mode. The trusted mode is the safer: the language will be run in a sandbox like environment and will not have direct access to system resources; on the other hand untrusted languages can escape the sandbox and operate directly on the hosting system. Developers and administrators have to carefully decide which language and context is allowed to be run in each database; by convention the untrusted version of a language is named as the language itself with a suffix ‘u’, so that for instance the trusted *plperl* language has the untrusted version identified as *plperlu*.

plperl

The Perl language support is provided by the *plperl* module, that can be installed using the port `databases/p5-postgresql-plperl`. To install the support for plperl in the example database it is required to execute the `CREATE LANGUAGE` statement (or the equivalent `createlang` shell command):

```
bsdmagdb=# CREATE LANGUAGE plperl;
```

Once the language has been installed into the database, it is possible to use it to create procedures and, from those, triggers. In the previous article a `compute_download_path` was defined to set the `download_path` field once the issuedon field is updated to a non-null value, and all the existent tuples are updated after a single `INSERT`. Using *plperl* the above function can be written as follows: Listing 1.

As readers can see the body of the trigger function is pure Perl. It is worth noting that trigger meta-data (such as the level or application context) are exported via the `$_TD` hash; moreover a Perl trigger function can apply changes returning the string “MODIFY” or abort changes returning the string “SKIP” when required. Most notably, Perl being a foreign language, the execution of SQL queries cannot be directly done, but special functions must be used to issue queries to the backend. Such functions are the

`spi_exec_query` family that, as readers can see, accept the SQL statement string and execute them.

Having defined the trigger function it is possible to associate it to triggers as in the previous example: Listing 2.

As readers can see, the `CREATE TRIGGER` command does not care about the language the trigger function has been declared into.

Consider now a little change to the example trigger that allows the function to copy a PDF file from a source directory (let say `~/pdf`) to a web server directory (let say `/www/downloads`): in order to do that the trigger function is going to use the *File::Copy* module to access the host file system. It is required that the procedure be declared in the "untrusted" version of the Perl language (*plperlu*), so drop the *plperl* support and add the *plperlu* one (see Box 2

Listing 1. *A trigger procedure written in plperl*

```
CREATE OR REPLACE FUNCTION  compute_download_path()
RETURNS trigger
AS
$BODY$

  # a variable to handle the default path
  my ($default_path);

  # check arguments
  if( $_TD->{argc} > 0 ){
     # array reference to the arguments
     @args = @{$_TD->{args}};
     $default_path = $args[0];
  }
  else{
     $default_path = 'http://bsdmag.org/download-demo/';
  }


  # update trigger, modify this single row
  if( $_TD->{event} eq "UPDATE" ){
    if( defined( $_TD->{new}->{issuedon} ) ){
        elog( LOG, "Applying a new download path" );
        $_TD->{new}->{download_path} = "'" . $default_
                    path . "BSD_" . $_TD->{new}->{id} .
                    ".pdf" . "'";
        elog( LOG, "New path is " . $_TD->{new}-
                    >{download_path} );
    }
    else{
      elog( LOG, "Deleting the download path");
      undef( $_TD->{new}->{download_path} );
    }

    # apply changes
    return "MODIFY";
  }

  # insert level?
  if( $_TD->{event} eq "INSERT" &&  $_TD->{level} eq
                    "STATEMENT" ){
     # here change all the rows calculating the download
                    path
     $query  = "UPDATE magazine SET download_path =
                    '$default_path' ";
     $query .= " || 'BSD_' || id || '.pdf'";
     $query .= " WHERE download_path IS NULL AND
                    issuedon IS NOT NULL";
     elog( LOG, "Perl query : $query " );
     spi_exec_query( $query );

      return;
  }

$BODY$
```

Listing 2. *Associating the plperl stored procedure to the SQL triggers*

```
CREATE TRIGGER tr_u_download_path
BEFORE UPDATE OF issuedon
ON magazine
FOR EACH ROW
EXECUTE PROCEDURE compute_download_path( 'http://
                    bsdmag.org/download-demo/' );


CREATE TRIGGER tr_i_download_path
AFTER INSERT
ON magazine
FOR EACH STATEMENT
EXECUTE PROCEDURE compute_download_path( 'http://
                    bsdmag.org/download-demo/' );
```

Listing 3. *Exploiting plperl-untrusted to access the local filesystem from within a trigger procedure*

```
bsdmagdb=# DROP LANGUAGE plperlu CASCADE;
bsdmagdb=# CREATE LANGUAGE plperlu;

CREATE OR REPLACE FUNCTION  compute_download_path()
RETURNS trigger
AS
$BODY$

  use File::Copy;

  # a variable to handle the default path
  # and the copy flag
  my ($default_path, $pdf_source_dir, $web_dir);

  # check arguments
  if( $_TD->{argc} > 0 ){
     # array reference to the arguments
     @args = @{$_TD->{args}};
     $default_path   = $args[0];
     $pdf_source_dir = $args[1];
     $web_dir        = $args[2];
  }
  else{
     $default_path = 'http://bsdmag.org/download-demo/';
  }


  # update trigger, modify this single row
  if( $_TD->{event} eq "UPDATE" ){
    if( defined( $_TD->{new}->{issuedon} ) ){
        elog( LOG, "Applying a new download path" );
        my $pdf_name = "BSD_" . $_TD->{new}->{id} .
                    ".pdf";
        my $pdf_path = "'" . $default_path . $pdf_name
                    . "'";

        elog( LOG, "New path is " . $_TD->{new}-
                    >{download_path} );

        # do I have to copy the file?
        if( defined($pdf_source_dir) ){
          my $pdf_dest = $web_dir . "/" . $pdf_name;
          if( ! -f $pdf_name ){
            my $pdf_source = $pdf_source_dir . "/" .
                    $pdf_name ;
            elog( LOG, "Copying file $pdf_source into
                    $pdf_dest ");
            copy( $pdf_source , $pdf_dest );
          }
        }
    }
    else{
      elog( LOG, "Deleting the download path");
      undef( $_TD->{new}->{download_path} );
    }

    # apply changes
    return "MODIFY";
  }


  # insert level?
  if( $_TD->{event} eq "INSERT" &&  $_TD->{level} eq
                    "STATEMENT" ){
     # here change all the rows calculating the download
                    path
     $query  = "UPDATE magazine SET download_path =
                    '$default_path' ";
     $query .= " || 'BSD_' || id || '.pdf'";
     $query .= " WHERE download_path IS NULL AND
                    issuedon IS NOT NULL";
     elog( LOG, "Perl query : $query " );
     spi_exec_query( $query );

      return;
  }

$BODY$
LANGUAGE plperlu;
```

for more information) and define the trigger function: Listing 3.

In the above function the trigger parameters are extended: there are two more parameters that provide the source and target directory for copying a PDF, so that the triggers are created as follows: Listing 4.

When the trigger is executed it is possible to see the copy of the file from the target directory to the destination directory: Listing 5.

An e-mail notification system

It is now possible to combine all the concepts expressed above and shown in the previous article to implement a simple e-mail notification system: each time a new issue of the *magazine* is placed into the magazine table an e-mail that notifies the availability of such issue is sent to a list of subscribed readers. The first step is to create and populate a table that will contain the readers information (e.g., name and e-mail): Listing 6.

Listing 4. *Associating the improved procedure to the SQL triggers*

```
CREATE TRIGGER tr_u_download_path
BEFORE UPDATE OF issuedon
ON magazine
FOR EACH ROW
EXECUTE PROCEDURE compute_download_path( 'http://bsdmag.org/download-demo/', '~/pdf/', '/www/downloads' );
AFTER INSERT
ON magazine
FOR EACH STATEMENT
EXECUTE PROCEDURE compute_download_path( 'http://bsdmag.org/download-demo/' , '~/pdf/', '/www/downloads');
```

Listing 5. *Firing the trigger execution*

```
bsdmagdb=# UPDATE magazine SET issuedon = '01-01-2010'::text::date WHERE title = 'Nessus and security';
LOG:  Applying a new download path
CONTEXT:  PL/Perl function "compute_download_path"
LOG:  New path is 'http://bsdmag.org/download-demo/BSD_2012-03.pdf'
CONTEXT:  PL/Perl function "compute_download_path"
LOG:  Copying file ~/pdf//BSD_2012-03.pdf into ~/www/pdf/BSD_2012-03.pdf
CONTEXT:  PL/Perl function "compute_download_path"
UPDATE 1
```

Listing 6. *Creating a small readers-archive*

```
CREATE TABLE readers(
   pk SERIAL NOT NULL,
   name  text,
   email text,
   PRIMARY KEY(pk),
   UNIQUE(email)
);

INSERT INTO readers(name, email) VALUES('Luca Ferrari', 'lf@fakemail.com');
INSERT INTO readers(name, email) VALUES('Ritchie Root', 'rr@fakemail.com');
ALTER TABLE magazine ADD COLUMN notified_readers integer DEFAULT 0;
```

Listing 7. *A plperlu procedure that will notify readers via e-mail*

```
CREATE OR REPLACE FUNCTION notify_readers( integer )
                    RETURNS integer
    LANGUAGE plperlu
    AS $_$

    use Mail::Sendmail;
    use MIME::Base64;
    use MIME::QuotedPrint;

    my ($issuepk)    = @_;
    my $sent_emails = 0;
    my $query        = "SELECT title, download_path FROM
                    magazine WHERE pk = $issuepk";
    my $boundary     = "===BSDMAG===";
    my $sent         = 0;
    my ($result_set, $email_text, $name, $current_row,
                    $email, %mail);
    my ($download_path, $title);

    elog( LOG, "Extracting the issue data via the query
                    $query\n");

    $current_row    = $result_set->{rows}[ 0 ];

    $title          = $current_row->{"title"};
    $download_path = $current_row->{"download_path"};
    elog( LOG, "Magazine Issue: $title at $download_path
                    \n" );

    $query       = "SELECT email, name FROM readers";
    elog( LOG, "Extracting all readers via the query
                    $query\n");
    $result_set = spi_exec_query( $query );

    $num_rows    = $result_set->{processed};
    elog( LOG, "Found $num_rows readers\n" );

    # iterate on each reader
    for( $i = 0; $i < $num_rows; $i++ ){
       $current_row = $result_set->{rows}[ $i ];
       $name        = $current_row->{"name"};
       $email       = $current_row->{"email"};
      # build the e-mail
       %mail = ( From     => 'postgres@bsdmag.org',
                 To       => $email,
                 Subject  => 'New BSD Magazine Issue!'
               );
       $mail{smtp} = 'localhost';
       $mail{body} = << "END_OF_BODY";
$boundary--
Content-Type: text/plain; charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable
Dear $name,
there is a new issue of BSD Magazine available for
download at the URL $download_path
so please check it out!
$boundary----
END_OF_BODY

       sendmail( %mail ) or warn( $Mail::Sendmail::
                    error) ;
       $sent++;
    }
    return $sent;
    $_$;
```

Listing 8. *Executing the notify_readers procedure for a specific user*

```
bsdmagdb=# select notify_readers( 2 );
LOG:  Extracting the issue data via the query SELECT
                    title, download_path FROM magazine
                    WHERE pk = 2
CONTEXT:  PL/Perl function "notify_readers"
LOG:  Magazine Issue: Rolling Your Own Kernel at http:
                    //bsdmag.org/download-demo/BSD_2011-
                    12.pdf
CONTEXT:  PL/Perl function "notify_readers"
LOG:  Extracting all readers via the query SELECT email,
                    name FROM readers
CONTEXT:  PL/Perl function "notify_readers"
LOG:  Found 2 readers

LOG:  Notifying reader Luca Ferrari at email
                    lf@fakemail.com
CONTEXT:  PL/Perl function "notify_readers"
LOG:  Notifying reader Ritchie Root at email
                    rr@fakemail.com
CONTEXT:  PL/Perl function "notify_readers"
 notify_readers
----------------
              2
```

The *magazine* table is also altered with a new column, `notified_readers`, that will track how many e-mail has been sent to notify the new issued paper. Now it is time to write a function that will accept an identifier for a magazine issue (e.g., the primary key) and will iterate over all the tuples in the *readers* table to extract every e-mail address and name in order to compose a customized e-mail text and send it. The following is the code of a *plperl* function that does the e-mail notification using the *Mail::Sendmail* and *MIME* modules: Listing 7.

As readers can see, the function is quite simple. It first performs the query against the *magazine* table to retrieve the tuple identified by the specified primary key. The tuple data is stored in the `$result_set` hash with the *rows* key, which contains an array of hashes each one identified with the name of the column. After having extracted the information about the magazine to notify, a cycle against all the tuple of the *readers* table is performed. Within such cycle a customized e-mail message is built and sent via a defined SMTP server. At the end the function returns the number of e-mail messages sent, that is the number of readers it tried to notify.

Executing the notification function is really straightforward: Listing 8.

Having defined the function to notify readers, it is possible to implement the logic that will fire the function for a newly issued *magazine*. The easiest way is to "attach" such a function to a trigger on the magazine table, so it is possible to change the trigger that computes the download path as follows: Listing 9.

As readers can see, if the `download_path` has been successfully computed, then the trigger executes the e-mail notification procedure `notify_readers` specifying the primary key (*pk*) of the tuple. It is then possible to

Listing 9. *Embedding the e-mail notification into the trigger procedure*

```
CREATE OR REPLACE FUNCTION compute_download_path()
RETURNS trigger AS
$BODY$
DECLARE
        default_path   text;
BEGIN

  -- check if the trigger has a path as argument,
  -- otherwise use a default path
  IF TG_NARGS > 0 THEN
     -- first argument is the path
     default_path := TG_ARGV[ 0 ];
     RAISE LOG 'Using a trigger-level path %', default_
                  path;
  ELSE
    -- a default hard coded path
    default_path :=  'http://bsdmag.org/download-demo/';
  END IF;

  -- print an LOG message
  RAISE LOG 'Trigger % executing for % event', TG_NAME,
                  TG_OP;


  -- if executing for a single column then compute the
                  path
  IF ( TG_OP = 'UPDATE' OR TG_OP = 'INSERT' )  THEN
     IF NEW.issuedon IS NOT NULL THEN
        NEW.download_path := default_path || 'BSD_' ||
                  NEW.id || '.pdf';
        RAISE LOG 'Computed download for issue % path is
                  %', NEW.title, NEW.download_path;
     ELSE
        RAISE LOG 'Removing the download path for issue
                  %', NEW.title;
        NEW.download_path    := NULL;
        NEW.notified_readers := 0;
     END IF;


     -- if here and the download path has been computed
     -- then notify readers
     IF NEW.download_path IS NOT NULL THEN
        SELECT notify_readers( NEW.pk )
        INTO   NEW.notified_readers;
        RAISE  LOG 'Notified readers %', NEW.notified_
                  readers;
     END IF;

     -- suppose this is a row trigger
     RETURN NEW;

  END IF;


END;
$BODY$
LANGUAGE plpgsql VOLATILE;
```

associate the above trigger function after the insert and update events of each row in the magazine table:

```
CREATE TRIGGER tr_download_path
AFTER INSERT OR UPDATE OF issuedon
ON magazine
FOR EACH ROW
EXECUTE PROCEDURE
compute_download_path( 'http://bsdmag.org/download-demo/' );
```

Listing 10 shows the execution of the trigger when a new *magazine* issue is inserted into the magazine table; it is possible to see all the messages of the trigger and of the `notify_readers` plperl procedure that iterates over the registered readers.

While the above described trigger could be an easy way to implement an automatic notification system, it is worth noting that `notify_readers` is not a "trigger-friendly" procedure. This means that if the procedure requires a long amount of time to complete its functionality, the trigger (and consequently the transaction) will have to wait for the procedure to complete. Moreover, if the procedure fails, the trigger will make the transaction to fail too. For this reason it is a better idea to untie the *notify_readers* function from the trigger on the *magazine* table and to execute it as a *periodic* or *cron* script. In order to do this, it is required first to be able to asynchronously identify issues for which no notifications have been sent, and to avoid notifying readers multiple times for the same issue. A simple and raw solution is to place a dummy value, let's say *-1*, into the `notified_readers` column of the magazine table each time a new row (that has the `download_path`) is added. The `compute_download_path` trigger function changes a single row and is associated this time as a "before" trigger to override the default value of 0 associated over the `notified_readers` column (it is also possible to drop such constraint), as shown in Listing 11. Then a shell script must be built

Listing 10. *Inserting new issue data and notifying the readers via a trigger*

```
bsdmagdb=# INSERT INTO magazine(id, month, issuedon,
                    title)
values('2008-01',1,'01/01/2008'::text::date, 'A very old
                    issue');
LOG:  Using a trigger-level path http://bsdmag.org/
                    download-demo/
LOG:  Trigger tr_download_path executing for INSERT
                    event
LOG:  Computed download for issue A very old issue path
                    is http://bsdmag.org/download-demo/
                    BSD_2008-01.pdf
LOG:  Extracting the issue data via the query SELECT
                    title, download_path FROM magazine
                    WHERE pk = 4157355

CONTEXT:  PL/Perl function "notify_readers"
SQL statement "SELECT notify_readers( NEW.pk )"
PL/pgSQL function "compute_download_path" line 36 at SQL
                    statement
LOG:  Magazine Issue: A very old issue at

CONTEXT:  PL/Perl function "notify_readers"
SQL statement "SELECT notify_readers( NEW.pk )"
PL/pgSQL function "compute_download_path" line 36 at SQL
                    statement
LOG:  Extracting all readers via the query SELECT email,
                    name FROM readers

CONTEXT:  PL/Perl function "notify_readers"
SQL statement "SELECT notify_readers( NEW.pk )"
PL/pgSQL function "compute_download_path" line 36 at SQL
                    statement
LOG:  Found 2 readers

CONTEXT:  PL/Perl function "notify_readers"
SQL statement "SELECT notify_readers( NEW.pk )"
PL/pgSQL function "compute_download_path" line 36 at SQL
                    statement
LOG:  Notifying reader Luca Ferrari at email
                    lf@fakemail.com

CONTEXT:  PL/Perl function "notify_readers"
SQL statement "SELECT notify_readers( NEW.pk )"
PL/pgSQL function "compute_download_path" line 36 at SQL
                    statement
LOG:  Notifying reader Ritchie Root at email
                    rr@fakemail.com

CONTEXT:  PL/Perl function "notify_readers"
SQL statement "SELECT notify_readers( NEW.pk )"
PL/pgSQL function "compute_download_path" line 36 at SQL
                    statement
LOG:  Notified readers 2
INSERT 0 1
```

to connect to the backend and execute the notification procedure over each tuple marked with a `notified_readers` value of *-1*. To make things more portable, a wrapping stored procedure is built (see Listing 12) to encapsulate the execution logic of the notification. The procedure `notify_readers_new_issue` (see Listing 12) iterates over each *magazine* tuple not yet notified and executes the `notify_readers` procedure. It is worth noting that, since a dynamically built query must be executed and its result is discarded, the *execute* statement is used.

Having defined the wrapping procedure, it does suffice to create a simple shell script as follows and to launch it via your favourite scheduler (*cron*, *periodic*, *at*, etc.):

```
#!/bin/sh

psql -U bsdmag -c „SET client_min_messages TO LOG; SELECT
                  notify_readers_new_issue(); „ bsdmagdb
```

Again, please note that the above implementation is not a very complex one due to both space limitations and didactic purposes; however it does suffice to illustrate how easy is to implement even a complex and articulated business logic directly into the PostgreSQL server side.

Inter-Process Communication: Listen/Notify

PostgreSQL embeds a simple pair of primitives for IPC: *listen* and its opposite *notify*. The idea is that a process can deliver an event with a specific content (payload) that other processes can receive and consume. The developer has to carefully choose the "channel" onto which delivering events, since its name must be unique and must be used on the counterpart listening processes.

Client applications can emit an event using the *notify* primitive through the connection driver, but usually it is better to embed the notification mechanism directly at

Listing 11. *The modified trigger function that places marks issues yet to be notified*

```
CREATE OR REPLACE FUNCTION compute_download_path()
RETURNS trigger AS
$BODY$
DECLARE
        default_path   text;
BEGIN

  -- check if the trigger has a path as argument,
  -- otherwise use a default path
  IF TG_NARGS > 0 THEN
     -- first argument is the path
     default_path := TG_ARGV[ 0 ];
     RAISE LOG 'Using a trigger-level path %', default_
                  path;
  ELSE
    -- a default hard coded path
    default_path :=  'http://bsdmag.org/download-demo/';
  END IF;

  -- print an info message
  RAISE LOG 'Trigger % executing for % event', TG_NAME,
                  TG_OP;



  -- if executing for a single column then compute the
                  path
  IF ( TG_OP = 'UPDATE' OR TG_OP = 'INSERT' )  THEN
     NEW.notified_readers := -1;

     IF NEW.issuedon IS NOT NULL THEN
        NEW.download_path := default_path || 'BSD_' ||
                  NEW.id || '.pdf';
        RAISE LOG 'Computed download for issue % path is
                  %', NEW.title, NEW.download_path;
     ELSE
        RAISE LOG 'Removing the download path for issue
                  %', NEW.title;
        NEW.download_path    := NULL;
        NEW.notified_readers := 0;
     END IF;



     -- suppose this is a row trigger
     RETURN NEW;

  END IF;


END;
$BODY$
LANGUAGE plpgsql VOLATILE;

CREATE TRIGGER tr_download_path
BEFORE INSERT OR UPDATE OF issuedon
ON magazine

EXECUTE PROCEDURE compute_download_path( 'http://
                  bsdmag.org/download-demo/' );
```

Listing 12. *A stored procedure that performs the call to the notification procedure*

```
CREATE OR REPLACE FUNCTION notify_readers_new_issue()
RETURNS void AS
$BODY$
DECLARE
        current_magazine        magazine%rowtype;
BEGIN

        FOR current_magazine IN SELECT * FROM magazine
                                WHERE notified_readers = -1
                                LOOP
                RAISE LOG 'Notification for issue %', current_magazine.title;
                EXECUTE 'SELECT notify_readers(' ||  current_magazine.pk ||' );';
        END LOOP;


END;
$BODY$
LANGUAGE plpgsql VOLATILE;
```

Listing 13. *A simple event listener written in Perl*

```
#!/usr/bin/env perl

use DBI;
my $database = "dbi:Pg:dbname=bsdmagdb";
my $username = 'bsdmag';
my $connection =  DBI->connect( $database, $username, '' ) || undef();

my $channel = "delete_channel";
print "\nListening on channel $channel\n";
$connection->do( "LISTEN $channel" );

for( my $i = 60; $i > 0; $i-- ){
    print "Waiting for still $i seconds...\n";
    sleep 1;

    # get the event back
    while( my $event = $connection->func("pg_notifies") ){
       if( defined($event) ) {
             my ($eventName, $pid, $payload) = @$event;
             print "Event <$eventName> received from process PID  <$pid> with payload <$payload>\n";
   }
    }
}
```

Box 1. plperl and pleperlu in the same database

Depending on the Perl installation it is possible that the database does not allow the usage of both *pleperl and plperlu*, claiming that another Perl interpreter cannot be allocated. This is due to security reasons: to avoid a privilege escalation from a *plperlu* code to a *plperl* one, the two languages must run over two different instances of Perl virtual machine. However, the virtual machine process is launched from a backend process, so the situation is that a single backend process must be able to launch two different Perl interpreters. There are two solutions to the problem: the first and simplest one is to drop the *plperl* language support and install only the *plperlu* into the database. This means that all your Perl code will run in untrusted mode, and can be a security risk. The other solution, the better one, is to recompile the Perl interpreter with the *usemultiplicty* flag that allows Perl to use different interpreters within the same process. In this way, only the code that really need to be run in untrusted mode will run in *plperlu*, while the other code can still run in trusted *plperl*.

Activating *usemultiplicity* in FreeBSD is simple and can be done at the configuration step of the port lang/perl5.10 checking the *multiplicity* checkbox. You can also ensure that the configuration is fine runing a `showconfig`:

```
# make showconfig
===> The following configuration options are available
                    for perl-threaded-5.10.1_6:
     ...
     MULTIPLICITY=on „Use multiplicity”
     ...
===> Use ‘make config’ to modify these settings
```

Please note that could be required to remove the old Perl installation via `pkg_delete(1)`.

the server-side, for instance within a rule. In this way the event will always be emitted, even if the client application crashes or has a problem. While the event notification is almost standard across languages and applications, and can be done either via the `NOTIFY` statement or the `pg_notify` function, the event listening depends on the client API used to connect to the database, even if PostgreSQL provides the `LISTEN` statement.

Listing 14. *A logger for „deletion” event coming out from PostgreSQL*

```
#!/usr/bin/env perl
use DBI;

my $database = "dbi:Pg:dbname=bsdmagdb";
my $username = 'bsdmag';

my $connection =  DBI->connect( $database, $username, ''
                    ) || undef();
my $channel = "delete_channel";
print "\nListening on channel $channel\n";
$connection->do( "LISTEN $channel" );
open( $LOG_FILE, ">>", "/tmp/deletion.log" ) ||
                    croack("Cannot create log file\n$!\
                    n");



while( 1 ){
    sleep 10;

    # get the event back
    while( my $event = $connection->func("pg_notifies") ){
    if( defined($event) ) {
        my ($eventName, $pid, $payload) = @$event;
        if( defined( $payload ) && $payload =~ /(.*)\
                    #(.*)\#/ ){

            # get the username and client ip address of
                    the notifier process
            $sql  = "SELECT usename, client_addr,
                    client_hostname";
            $sql .= " FROM pg_stat_activity ";
            $sql .= " WHERE procpid = $pid; ";
            $resultset_arrayref = $connection-
                    >selectall_arrayref( $sql );
            my $username = $resultset_arrayref->[0][0];
            my $ip       = $resultset_arrayref->[0][1];
            my $hostname = $resultset_arrayref->[0][2];

            print $LOG_FILE "#### DELETION EVENT ####\n";
            print $LOG_FILE "Backend process $pid deleted
                    the magazine issue titled $2\n";
            print $LOG_FILE "\tUsername $username from
                    client $ip ($hostname)\n";
        }

    }
    }
    }

close( $LOG_FILE );
```

Box 2. The importance of the "pl" in PostgreSQL languages

Having PostgreSQL to support a lot of different programming languages is really useful and allows the reuse of a lot of code and a lot of libraries. Moreover, having PostgreSQL to support even object oriented languages such as Perl or Java makes it even more attractive. However it is worth noting the presence of that "pl" prefix in front of each language: such "pl" stands for "Procedural Language". It essentially means that it does not matter how smart a developer is writing excellent OOP Java code, PostgreSQL will manage it as a procedural language. That does not mean that OOP is forbidden, and developers can have their own OOP libraries, but the entry point for PostgreSQL is always a function.

Box 3. DBI::Pg and DBI::PgPP

Perl provides two main DBI implementation for PostgreSQL: *Pg* and *PgPP*. The former is the first implementation made available and requires *libpq*, the library for PostgreSQL C clients, to be installed on the system. The latter is a pure Perl implementation and does not require *libpq* to be installed, allowing the module to directly handle and parse the networking messages from a to a backend process.

On The Web

- PostgreSQL official Web Site: *http://www.postgresql.org*
- ITPUG official Web Site: *http://www.itpug.org*
- PostrgeSQL plpgsql Documentation: *http://www.postgresql.org/docs/current/static/plpgsql-statements.html*
- PostgreSQL Rule System documentation: *http://www.postgresql.org/docs/current/static/rules.html*
- PostgreSQL Triggers Documentation: *http://www.postgresql.org/docs/current/static/triggers.html*
- GitHub Repository containing the source code of the examples: *https://github.com/fluca1978/fluca-pg-utils*

In order to demonstrate the usage of the IPC primitives consider this scenario: each time a new issue is going to be deleted a notification will be sent to a client application that will log the event in order to report it to an administrator. The first step is to define a rule that will perform the event notification along with the deletion action:

```
CREATE OR REPLACE RULE r_delete_magazine
AS ON DELETE TO magazine
DO ALSO
NOTIFY delete_channel, 'Deletion of a tuple';
```

As readers can see, once a deletion will be performed, also a notification over the `delete_channel` event channel will be sent: an event with the custom content (payload) of "Deletion of a tuple" will be notified to listeners. It is possible to test such notification mechanism even in the `psql` terminal issuing a `LISTEN` statement for the chosen channel `delete_channel`:

```
bsdmagdb=# LISTEN delete_channel;
bsdmagdb=# DELETE FROM magazine WHERE id = '2007-01';
Asynchronous notification „delete_channel" with payload
„Deletion of a tuple" received from server process with
                    PID 1357.
```

Before continuing to define the client application that will handle and consume the above events, it is worth noting that the event issued by the rule does not contain a lot of information about the tuple that is going to be deleted. Luckily PostgreSQL provides the function `pg_notify` that can be used when dealing with run-time build payload or event channel naming, so that the rule can be changed to the following:

```
CREATE OR REPLACE RULE r_delete_magazine
AS ON DELETE TO magazine
DO ALSO
SELECT pg_notify( 'delete_channel', 'Deletion of the tuple
                    titled: #' || OLD.title || '#');
```

so that the event will contain a more complete payload, as shown in the following deletion example:

```
bsdmagdb=# DELETE FROM magazine WHERE id = '2007-01';
Asynchronous notification „delete_channel" with payload
„Deletion of the tuple titled: #A very old issue#" received
from server process with PID 1357.
```

Now that the notification engine is in place, it is time to write a simple client application that can consume the incoming events; for this purposes a Perl script will be used. Perl uses the module *DBI::Pg* (or *BDI:PgPP*) to connect to a PostgreSQL database; the module can be installed via the ports tree or the CPAN shell. First of all let's see a simple script that will report on standard output each received event: Listing 13.

The above script connects to the database and issues a `LISTEN` for the specified event channel (`delete_channel`); after that, the script calls the special DBI function `pg_notifies` every second, which provides a single event out of the listening channel, to print its information (payload, sender process id and event name). Being able to process the event information, including the payload, allows the script to implement the desired logging; moreover the script can also connect back to the database and query the `pg_stat_activity` view to get information about which user did the deletion. The whole logger implementation is reported in Listing 14. Each time a deletion event is notified to the client process, a log entry like the following is added to the `/tmp/deletion.log` file:

```
#### DELETION EVENT ####
Backend process 3022 deleted the magazine issue titled
                    FreeBSD: Get Up To Date
        Username bsdmag from client 192.168.200.1 (flucabsd)
```

Of course, the logger shown in this example is really simple and not meant to be solid-as-a-rock. Again, the idea is to show what PostgreSQL can do and how easy it can be to develop complex database applications using server-side features. It is worth noting that the PostgreSQL IPC has some limitations, most notably the use of a text payload that makes difficult the manipulation of non-text data. Moreover, the event queue can be filled with unread messages, even if the queue can keep up to 8GB of events. Last but not least, the IPC is transaction-boundary aware, and therefore a `NOTIFY` will be issued at the commit of the running transaction.

As an exercise for the readers, it is possible to convert the e-mail notification system using IPC so that each time a new tuple is made available an event is notified to an external mailer process that sends the e-mail.

Summary and Coming Next

This article completed the glance at the server-side programming in PostgreSQL, with particular regard to the use of foreign languages (Perl) and the IPC mechanism implemented via listen/notify. In the next article the data partitioning capabilities will be shown.

LUCA FERRARI

Luca Ferrari lives in Italy with his wife and son. He is an Adjunct Professor at Nipissing University, Canada, a co-founder and the vice-president of the Italian PostgreSQL Users' Group (ITPUG). He simply loves the Open Source culture and refuses to log-in to non-Unix systems. He can be reached on line at http://fluca1978.blogspot.com

Synchronization Problems or:

How I Learned to Stop Worrying and Love the Sleep Mutex

This article addresses the problem of data and state corruption caused by concurrent threads.

What you will learn…

- An in-depth understanding of synchronization problems
- The negative effects of synchronization problems
- One solution to synchronization problems (hint: I'm going to focus on sleep mutexes)

What you should know…

- The C programming language
- Rudimentary FreeBSD kernel module programming
- If you lack the abovementioned prerequisites, you should still be able to read this article (you just might not understand the code listings)

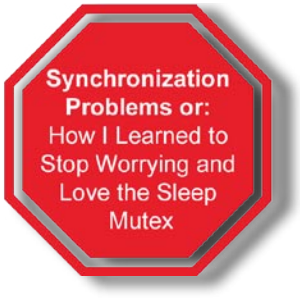
When two or more threads executing on different processors simultaneously manipulate the same data structure, that structure can be corrupted. Fortunately, FreeBSD contains multiple solutions to this problem. Before I describe these solutions, you'll require an in-depth understanding of the abovementioned problem; formally known as a synchronization problem.

Listing 1 shows a function named `race_new()` that adds a structure (at line 18) to a doubly linked list named `race_list`. Every structure contains a unique unit number (calculated at lines 07-11).

Listing 2 shows an ioctl routine named `race_ioctl()` that calls `race_new()` (at line 10) to add a structure to `race_list`.

For completeness sake, Listing 3 shows the kernel module that Listings 1 and 2 are from.

Listing 4 shows a command-line utility designed to invoke `race_ioctl()` in order to add a structure to `race_list`.



Listing 1. *race_new() Function*

```
01 static struct race_softc *
02 race_new(void)
03 {
04         struct race_softc *sc;
05         int unit, max = -1;
06
07         LIST_FOREACH(sc, &race_list, list) {
08                 if (sc->unit > max)
09                         max = sc->unit;
10         }
11         unit = max + 1;
12
13         sc = malloc(sizeof(struct race_softc),
                    M_RACE, M_NOWAIT | M_ZERO);
14         if (sc == NULL)
15                 return (NULL);
16
17         sc->unit = unit;
18         LIST_INSERT_HEAD(&race_list, sc, list);
19
20         return (sc);
21 }
```

Listing 2. *race_ioctl() Function*

```
01 static int
02 race_ioctl(struct cdev *dev, u_long cmd, caddr_t
                  data, int fflag,
03     struct thread *td)
04 {
05         struct race_softc *sc;
06         int error = 0;
07
08         switch (cmd) {
09         case RACE_IOC_ATTACH:
10                 sc = race_new();
11                 if (sc == NULL) {
12                         error = ENOMEM;
13                         break;
14                 }
15                 *(int *)data = sc->unit;
16                 break;
17         default:
18                 error = ENOTTY;
19                 break;
20         }
21
22         return (error);
23 }
```

Listing 3a. *race.c*

```
#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/uio.h>
#include <sys/malloc.h>
#include <sys/ioccom.h>
#include <sys/queue.h>

#define RACE_NAME               "race"
#define RACE_IOC_ATTACH         _IOR('R', 0, int)

static MALLOC_DEFINE(M_RACE, RACE_NAME, "race object");

struct race_softc {
        LIST_ENTRY(race_softc) list;
        int unit;
};

static LIST_HEAD(, race_softc) race_list =
    LIST_HEAD_INITIALIZER(&race_list);

static struct race_softc *      race_new(void);
static d_ioctl_t                race_ioctl;

static struct cdevsw race_cdevsw = {
        .d_version =    D_VERSION,
        .d_ioctl =      race_ioctl,
        .d_name =       RACE_NAME
};

static struct cdev *race_dev;

static int
race_ioctl(struct cdev *dev, u_long cmd, caddr_t data,
    int fflag, struct thread *td)
{
/* See Listing 2 for function definition. */
...
}

static struct race_softc *
race_new(void)
{
/* See Listing 1 for function definition. */
...
}

static int
race_modevent(module_t mod __unused, int event,
    void *arg __unused)
{
        int error = 0;

        switch (event) {
        case MOD_LOAD:
                race_dev = make_dev(&race_cdevsw, 0,
                    UID_ROOT, GID_WHEEL, 0600,
                    RACE_NAME);
                uprintf("Race driver loaded.\n");
                break;
        case MOD_UNLOAD:
                destroy_dev(race_dev);
                uprintf("Race driver unloaded.\n");
                break;
        default:
                error = EOPNOTSUPP;
                break;
```

Listing 3b. *race.c*

```
        }

        return (error);
}

DEV_MODULE(race, race_modevent, NULL);
```

Listing 4. *race_config.c*

```
#include <sys/types.h>
#include <sys/ioctl.h>

#include <err.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define RACE_NAME               "race"
#define RACE_IOC_ATTACH         _IOR('R', 0, int)

static enum {UNSET, ATTACH} action = UNSET;

/*
 * The usage statement: race_config -a
 */
static void
usage()
{
        fprintf(stderr, "usage: race_config -a\n");
        exit(1);
}

/*
 * This program manages the doubly linked list found in
 * /dev/race. It allows you to add an item to the list.
 */
int
main(int argc, char *argv[])
{
        int ch, fd, i, unit;

        /*
         * Parse the command line argument list to
         * determine the correct course of action.
         *
         *    -a:      add an item.
         */
        while ((ch = getopt(argc, argv, "a")) != -1)
                switch (ch) {
                case 'a':
                        if (action != UNSET)
                                usage();
                        action = ATTACH;
                        break;
                default:
                        usage();
                }

        /*
         * Perform the chosen action.
         */
        if (action == ATTACH) {
                fd = open("/dev/" RACE_NAME, O_RDWR);
                if (fd < 0)
                        err(1, "open(/dev/%s)",
                            RACE_NAME);

                i = ioctl(fd, RACE_IOC_ATTACH, &unit);
                if (i < 0)
                        err(1, "ioctl(/dev/%s)",
                            RACE_NAME);
                printf("unit: %d\n", unit);

                close (fd);
        } else
                usage();

        return (0);
}
```

Listing 5a. *race_mtx.c*

```
001 #include <sys/param.h>
002 #include <sys/module.h>
003 #include <sys/kernel.h>
004 #include <sys/systm.h>
005
006 #include <sys/conf.h>
007 #include <sys/uio.h>
008 #include <sys/malloc.h>
009 #include <sys/ioccom.h>
010 #include <sys/queue.h>
011 #include <sys/lock.h>
012 #include <sys/mutex.h>
013
014 #define RACE_NAME               "race"
015 #define RACE_IOC_ATTACH         _IOR('R', 0, int)
016
017 static MALLOC_DEFINE(M_RACE, RACE_NAME, "race
                object");
018
019 struct race_softc {
020         LIST_ENTRY(race_softc) list;
021         int unit;
022 };
023
024 static LIST_HEAD(, race_softc) race_list =
025     LIST_HEAD_INITIALIZER(&race_list);
026
027 static struct mtx race_mtx;
028
029 static struct race_softc *      race_new(void);
030 static d_ioctl_t                race_ioctl_mtx;
031 static d_ioctl_t                race_ioctl;
032
033 static struct cdevsw race_cdevsw = {
034         .d_version =    D_VERSION,
035         .d_ioctl =      race_ioctl_mtx,
036         .d_name =       RACE_NAME
037 };
038
039 static struct cdev *race_dev;
040
041 static int
042 race_ioctl_mtx(struct cdev *dev, u_long cmd, caddr_t
                data, int fflag,
043     struct thread *td)
044 {
045         int error;
046
047         mtx_lock(&race_mtx);
048         error = race_ioctl(dev, cmd, data, fflag,
                td);
049         mtx_unlock(&race_mtx);
050
051         return (error);
052 }
053
054 static int
055 race_ioctl(struct cdev *dev, u_long cmd, caddr_t
                data, int fflag,
056     struct thread *td)
057 {
058 /* See Listing 2 for function definition. */
059 ...
060 }
061
062 static struct race_softc *
063 race_new(void)
064 {
065 /* See Listing 1 for function definition. */
066 ...
067 }
068
069 static int
070 race_modevent(module_t mod __unused, int event, void
                *arg __unused)
071 {
072         int error = 0;
073         struct race_softc *sc, *sc_temp;
074
075         switch (event) {
076         case MOD_LOAD:
077                 mtx_init(&race_mtx, "race config
                lock", NULL, MTX_DEF);
078                 race_dev = make_dev(&race_cdevsw, 0,
                UID_ROOT, GID_WHEEL,
079                     0600, RACE_NAME);
080                 uprintf("Race driver loaded.\n");
081                 break;
082         case MOD_UNLOAD:
083                 destroy_dev(race_dev);
084                 mtx_lock(&race_mtx);
085                 if (!LIST_EMPTY(&race_list)) {
086                         LIST_FOREACH_SAFE(sc, &race_
                list, list, sc_temp)
087                         {
088                                 LIST_REMOVE(sc, list);
```

If two threads execute Listing 4 simultaneously, this might occur:

```
$ sudo kldload ./race.ko
Password:
Race driver loaded.
$ cd ../race_config/
$ sudo ./race_config -a & sudo ./race_config -a &
[1] 1128
[2] 1129
$ unit: 0
unit: 0
```

As you can see, two structures ended up with the same "unique" unit number. Naturally, this is undesirable and should not occur. So what happened? Well, both threads checked `race_list` simultaneously, discovered that it was empty, and assigned `0` as the unit number. In other words, due to a particular sequence of events, an error occurred. This error is known as a race condition.

Preventing Race Conditions

Race conditions are prevented using locks. Locks, also known as synchronization primitives, are used to serialize the execution of two or more threads. For example, the abovementioned race condition is caused by concurrent access to `race_list` and can be prevented by using a lock to serialize access to `race_list`. Before a thread can access `race_list`, it must first acquire the `foo` lock. Only one thread can hold `foo` at a time. If a thread cannot acquire `foo`, it cannot access `race_list` and must wait for the current owner to relinquish `foo`. This protocol guarantees that at any moment in time only one thread can access `race_list`.

In FreeBSD there are numerous types of locks. The most commonly used lock is the sleep mutex (mutex is a portmanteau of mutual and exclusion). If a thread attempts to acquire a sleep mutex that is being held by another thread, it will context switch (that is, sleep) and wait for the sleep mutex to be released. Incidentally, the `foo` lock described previously is a sleep mutex.

Listing 5 shows how a sleep mutex can prevent the aforementioned race condition. It works by serializing the execution of `race_ioctl()`, which is the main source of concurrent access to `race_list`. A new function named `race_ioctl_mtx()` is defined as the ioctl routine (shown at line 35). This function begins by acquiring a sleep mutex (at line 47). Next, `race_ioctl()` is called (at line 48) and after it returns the sleep mutex is released (at line 49).

As you can see, it takes just one lock to serialize the execution of `race_ioctl()`. Essentially, this is what all locks do. They prevent multiple threads from simultaneously manipulating one or more objects.

Listing 5b. *race_mtx.c*

```
089                                     free(sc, M_RACE);
090                             }
091                     }
092                     mtx_unlock(&race_mtx);
093                     mtx_destroy(&race_mtx);
094                     uprintf("Race driver unloaded.\
                        n");
095                     break;
096             default:
097                     error = EOPNOTSUPP;
098                     break;
099             }
100
101             return (error);
102 }
103
104 DEV_MODULE(race, race_modevent, NULL);
```

Closing Words

If you take just one thing away from this article, it should be this: Whenever an object can be accessed by multiple threads, you *must* manage that access.

As an aside, one characteristic of race conditions is that they're hard to reproduce. Thus, the race condition was doctored in this article. That is, I caused the threads to context switch at key points in order to achieve the desired outcome. Under normal conditions, it would have taken me literally millions of attempts before that race condition would occur, and I didn't want to spend my time doing that.

JOSEPH KONG

The author of Designing BSD Rootkits (No Starch Press) and FreeBSD Device Drivers (No Starch Press), Joseph Kong dabbles in operating system design, reverse code engineering, and computer (in)security.

Great Specials

On FreeBSD & PC-BSD Merchandise

Give us a call & ask about our
SOFTWARE BUNDLES
1.925.240.6652

$39.95
FreeBSD 9.0 Jewel Case CD Set
or FreeBSD 9.0 DVD

$29.95
PC-BSD 9.0 DVD

$49.95 Save a BUNDLE!
The PC-BSD 9.0 Users Handbook
PC-BSD 9.0 DVD

$99.95
The FreeBSD CD **or** DVD Bundle

Inside each CD/DVD Bundle, you'll find:
FreeBSD Handbook, 3rd Edition
Users Guide FreeBSD Handbook, 3rd Edition, Admin Guide
FreeBSD 9.0 CD or DVD set
FreeBSD Toolkit DVD

Stylish Dress Attire
Look Your Professional Best

Comfy Hoodies
Stay Warm in Pullovers & Zip Ups

T-Shirts
Lots of Styles to Choose From

FreeBSD 9.0 Jewel Case CD/DVD $39.95

CD Set Contains:

- **Disc 1:** Installation Boot LiveCD (i386)
- **Disc 2:** Essential Packages Xorg, GNOME2 (i386)
- **Disc 3:** Installation Boot LiveCD (amd64)
- **Disc 4:** Essential Packages Xorg, GNOME2 (amd64)

FreeBSD 8.2 CD $39.95
FreeBSD 8.2 DVD $39.95

FreeBSD Subscriptions

Save time and $$$ by subscribing to regular updates of FreeBSD

FreeBSD Subscription, start with CD 9.0 ..... $29.95
FreeBSD Subscription, start with DVD 9.0 ..... $29.95
FreeBSD Subscription, start with CD 8.2 ..... $29.95
FreeBSD Subscription, start with DVD 8.2 ..... $29.95

PC-BSD 9.0 DVD (Isotope Edition)

PC-BSD 9.0 DVD $29.95
PC-BSD Subscription $19.95

The FreeBSD Handbook

The FreeBSD Handbook, Volume 1 (User Guide) $39.95
The FreeBSD Handbook, Volume 2 (Admin Guide) $39.95

The FreeBSD Handbook Specials

The FreeBSD Handbook, Volume 2 (Both Volumes) $59.95
The FreeBSD Handbook, Both Volumes & FreeBSD 9.0 ..... $79.95

PC-BSD 9.0 Users Handbook $24.95

BSD Magazine $11.99

The FreeBSD Toolkit DVD $39.95

FreeBSD Mousepad $10.00

FreeBSD & PCBSD Caps $20.00

BSD Daemon Horns $2.00

Bundle Specials!
Save $$$

Just Plain Fun
Mousepads & Novelty Horns

BSD Magazine
Available Monthly

FreeBSD Mall

For even **MORE** items
visit our website today!

www.FreeBSDMall.com

ZFS Madness with BEADM

Some time ago I found a good, reliable way of using and installing FreeBSD and described it in my Modern FreeBSD Install [1] [2] HOWTO. Now, more then a year later I come back with my experiences about that setup and a proposal of newer and a lot better way of doing it.

What you will learn…

- How to install FreeBSD in the most useful way.
- How to implement and use Boot Environments with beadm utility.
- The internals of the FreeBSD install process.
- How to make basic FreeBSD configuration after installation process.

What you should know…

- Knowledge about ZFS concepts would be useful.

Same as year ago, I assume that you would want to create fresh installation of FreeBSD using one or more hard disks, but also with (laptops) and without GELI based full disk encryption.

This guide was written when FreeBSD 9.0 and 8.3 were available and definitely works for 9.0, but I did not try all this on the older 8.3, if you find some issues on 8.3, let me know I will try to address them in this guide.

Earlier, I was not that confident about booting from the ZFS pool, but there is some very neat feature that made me think ZFS boot is now mandatory. If you just smiled, you know that I am thinking about *Boot Environments* feature from Illumos/Solaris systems.

In case you are not familiar with the *Boot Environments* feature, check the *Managing Boot Environments with Solaris 11 Express PDF* white paper [3]. Illumos/Solaris has the `beadm(1M)` [4] utility and while Philipp Wuensche wrote the manageBE script as replacement [5], it uses older style used at times when OpenSolaris (and SUN) were still having a great time.

I spent last couple of days writing an up-to-date replacement for FreeBSD compatible `beadm` utility (Listing 1.), and with some tweaks from today I just made it available at SourceForge [6] or *GitHub* [7] if you wish to test it. Currently its about 200 lines long, so it should be pretty simple to take a look at it. I tried to make it as compatible as possible with the 'upstream' version, along with some small improvements, it currently supports basic functions like list, create, destroy, activate and rename.

There are several subtle differences between mine implementation and Philipp's one, he defines and then relies upon ZFS property called `freebsd:boot-environment=1` for each boot environment, I do not set any other additional ZFS properties. There is already `org.freebsd:swap` property used for SWAP on FreeBSD, so we may use `org.freebsd:be` in the future, but is just a thought, right now its not used.

Listing 1. *beadm usage*

```
# beadm
usage:
  beadm subcommand cmd_options

  subcommands:

  beadm activate beName
  beadm create [-e nonActiveBe | beName@snapshot] beName
  beadm create beName@snapshot
  beadm destroy beName
  beadm destroy beName@snapshot
  beadm list
  beadm rename origBeName newBeName
```

My version also supports activating boot environments received with `zfs recv` command from other systems (it just updates appreciate `/boot/zfs/zpool.cache` file).

My implementation is also style compatible with current Illumos/Solaris `beadm(1M)` which is like the example below (Listing 2).

The boot environments are located in the same please as in Illumos/Solaris, at `pool/ROOT/environment` place.

Now you're Thinking with Portals [*]

[*] Reference to the Portal computer game from Valve.

The main purpose of the *Boot Environments* concept is to make all risky tasks harmless, to provide an easy way back from possible troubles. Think about upgrading the system to newer version, an update of 30+ installed packages to latest versions, testing software or various solutions before taking the final decision, and much more. All these tasks are now harmless thanks to the *Boot Environments*, but this is just the tip of the iceberg.

You can now move desired boot environment to other machine, physical or virtual and check how it will behave there, check hardware support on the other hardware for

Listing 2. *beadm command in action*

```
# beadm create -e default upgrade-test
Created successfully

# beadm list
BE           Active Mountpoint Space Policy Created
default      N    /       1.06M static 2012-02-03 15:08
upgrade-test R    -       560M static 2012-04-24 22:22
new          -    -       8K static 2012-04-24 23:40

# zfs list -r sys/ROOT
NAME                   USED  AVAIL  REFER  MOUNTPOINT
sys/ROOT               562M  8.15G   144K  none
sys/ROOT/default      1.48M  8.15G   558M  legacy
sys/ROOT/new             8K  8.15G   558M  none
sys/ROOT/upgrade-test  560M  8.15G   558M  none

# beadm activate default
Activated successfully

# beadm list
BE           Active Mountpoint Space Policy Created
default      NR   /       1.06M static 2012-02-03 15:08
upgrade-test -    -       560M static 2012-04-24 22:22
new          -    -       8K static 2012-04-24 23:40
```

example or make a painless hardware upgrade. You may also clone your desired boot environment and ... start it as a Jail for some more experiments or move your old physical server install into FreeBSD Jail because its not that heavily used anymore but it still have to be available.

Other good example may be just created server on your laptop inside VirtualBox virtual machine. After you finish the creation process and tests, you may move this boot environment to the real server and put it into production. Or even move it into VMware ESX/vSphere virtual machine and use it there. As you see the possibilities with *Boot Environments* are unlimited.

The Install Process

I created 3 possible schemes which should cover most demands, choose one and continue to the next step.

Server with Two Disks

I assume that this server has 2 disks and we will create ZFS mirror across them, so if any of them will be gone the

Listing 3. *Server with Two Disks install process*

```
1. Boot from the FreeBSD USB/DVD.
2. Select the 'Live CD' option.
3. login: root
4. # sh
5. # DISKS="ada0 ada1"
6. # for I in ${DISKS}; do
> NUMBER=$( echo ${I} | tr -c -d '0-9' )

> gpart create -s GPT ${I}
> gpart add -t freebsd-boot -l bootcode${NUMBER} -s
                    128k ${I}
> gpart add -t freebsd-zfs -l sys${NUMBER} ${I}
> gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i
                    1 ${I}
> done
7. # zpool create -f -o cachefile=/tmp/zpool.cache sys
                    mirror /dev/gpt/sys*
8. # zfs set mountpoint=none sys
9. # zfs set checksum=fletcher4 sys
10. # zfs set atime=off sys
11. # zfs create sys/ROOT
12. # zfs create -o mountpoint=/mnt sys/ROOT/default
13. # zpool set bootfs=sys/ROOT/default sys
14. # cd /usr/freebsd-dist/
15. # for I in base.txz kernel.txz; do
> tar --unlink -xvpJf ${I} -C /mnt
> done
16.# cp /tmp/zpool.cache /mnt/boot/zfs/
17.# cat << EOF >> /mnt/boot/loader.conf
> zfs_load=YES
> vfs.root.mountfrom="zfs:sys/ROOT/default"
> EOF
18.# cat << EOF >> /mnt/etc/rc.conf
> zfs_enable=YES
> EOF
19.# :> /mnt/etc/fstab
20.# zfs umount -a
21.# zfs set mountpoint=legacy sys/ROOT/default
22.# reboot
```

Listing 4. *Disk layout after Server with Two Disks install process*

```
# gpart show
=>      34  1048509  ada0  GPT  (512M)
        34      256     1  freebsd-boot  (128k)
       290  1048253     2  freebsd-zfs  (511M)

=>      34  1048509  ada1  GPT  (512M)
        34      256     1  freebsd-boot  (128k)
       290  1048253     2  freebsd-zfs  (511M)

# gpart list | grep label
   label: bootcode0
   label: sys0
   label: bootcode1
   label: sys1

# zpool status
  pool: sys
 state: ONLINE
 scan: none requested
config:

        NAME          STATE     READ WRITE CKSUM
        sys           ONLINE       0     0     0
          mirror-0    ONLINE       0     0     0
            gpt/sys0  ONLINE       0     0     0
            gpt/sys1  ONLINE       0     0     0

errors: No known data errors
```

system will still work as usual. I also assume that these disks are `ada0` and `ada1`. If you have SCSI/SAS drives there, they may be named `da0` and `da1` accordingly. The procedures below (Listing 3) will wipe all data on these disks, you have been warned.After these instructions and reboot we have these GPT partitions available, this example is on a 512MB disk (Listing 4).

Server with One Disk

If your server configuration has only one disk, lets assume its ada0, then you need different points 5. and 7. to make, use these instead of the ones above.

```
5. # DISKS="ada0"
7. # zpool create -f -o cachefile=/tmp/zpool.cache sys
                    /dev/gpt/sys*
```

All other steps are the same.

Road Warrior Laptop

The procedure is quite different for Laptop because we will use the full disk encryption mechanism provided by GELI and then setup the ZFS pool. Its not currently possible to boot off from the ZFS pool on top of encrypted GELI provider, so we will use setup similar to the Server with ... one but with additional `local` pool for `/home` and `/root` partitions. It will be password based and you will be asked to type-in that password at every boot. The install process is generally the same with new instructions added for the GELI encrypted `local` pool, I put them with *different color* to make the difference more visible. After these instructions and reboot we have these GPT partitions available, this example is on a 4GB disk (Listing 6).

Basic Setup after Install

- Login as `root` with empty password.
  ```
  login: root
  password: [ENTER]
  ```
- Create initial `snapshot` after install. `# zfs snapshot -r sys/ROOT/default@install`
- Set new `root` password. `# passwd`
- Set machine's `hostname`. `# echo hostname=hostname.domain.com >> /etc/rc.conf`
- Set proper `timezone`. `# tzsetup`
- Add some `swap` space.
  If you used the *Server with ...* type, then use this to add swap.
  ```
  # zfs create -V 1G -o org.freebsd:swap=on \
                    -o checksum=off \
                    -o sync=disabled \
                    -o primarycache=none \
                    -o secondarycache=none sys/swap
  # swapon /dev/zvol/sys/swap
  ```
  If you used the *Road Warrior Laptop* one, then use this one below, this way the swap space will also be encrypted.
  ```
  # zfs create -V 1G -o org.freebsd:swap=on \
                    -o checksum=off \
                    -o sync=disabled \
                    -o primarycache=none \
                    -o secondarycache=none local/swap
  # swapon /dev/zvol/local/swap
  ```
- Create `snapshot` called `configured` or `production`

After you configured your fresh FreeBSD system, added needed packages and services, create `snapshot` called `configured` or `production` so if you mess something, you can always go back in time to bring working configuration back.

```
# zfs snapshot -r sys/ROOT/default@configured
```

Enable Boot Environments

Here are some simple instructions on how to download and enable the `beadm` command line utility for *easy Boot Environments* administration.

```
# fetch -o /usr/sbin/beadm https://downloads.sourceforge.n
                    et/project/beadm/beadm
# chmod +x /usr/sbin/beadm
# rehash
# beadm list
BE      Active Mountpoint Space Policy Created
default NR     /          592M static 2012-04-25 02:03
```

Possible Usage Patters

Now we have a working ZFS only FreeBSD system, I will put some example here about what you now can do with this type of installation and of course the *Boot Environments* feature.

Create New Boot Environment Before Upgrade

- Create new environment from the current one. `# beadm create upgrade`
- Activate it. `# beadm activate upgrade`
- Reboot into it. `# shutdown -r now`
- Mess with it.

You are now free to do anything you like for or the upgrade process, but even if you break everything, you still have a working `default` working environment.

Listing 5. *Road Warrior Laptop install process*

```
1. Boot from the FreeBSD USB/DVD.
2. Select the 'Live CD' option.
3. login: root
4. # sh
5. # DISKS="ada0"
6. # for I in ${DISKS}; do
> NUMBER=$( echo ${I} | tr -c -d '0-9' )
> gpart destroy -F ${I}
> gpart create -s GPT ${I}
> gpart add -t freebsd-boot -l bootcode${NUMBER} -s
                    128k ${I}
> gpart add -t freebsd-zfs -l sys${NUMBER} -s 10G ${I}
> gpart add -t freebsd-zfs -l local${NUMBER} ${I}
> gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i
                    1 ${I}
> done
7. # zpool create -f -o cachefile=/tmp/zpool.cache sys
                    /dev/gpt/sys0
8. # zfs set mountpoint=none sys
9. # zfs set checksum=fletcher4 sys
10. # zfs set atime=off sys
11. # zfs create sys/ROOT
12. # zfs create -o mountpoint=/mnt sys/ROOT/default
13. # zpool set bootfs=sys/ROOT/default sys
14. # geli init -b -s 4096 -e AES-CBC -l 128 /dev/gpt/
                    local0
15. # geli attach /dev/gpt/local0
16. # zpool create -f -o cachefile=/tmp/zpool.cache
                    local /dev/gpt/local0.eli
17. # zfs set mountpoint=none local
18. # zfs set checksum=fletcher4 local
19. # zfs set atime=off local
20. # zfs create local/home
21.# zfs create -o mountpoint=/mnt/root local/root
22.# cd /usr/freebsd-dist/
23.# for I in base.txz kernel.txz; do
> tar --unlink -xvpJf ${I} -C /mnt
> done
24.# cp /tmp/zpool.cache /mnt/boot/zfs/
25.# cat << EOF >> /mnt/boot/loader.conf
> zfs_load=YES
> geom_eli_load=YES
> vfs.root.mountfrom="zfs:sys/ROOT/default"
> EOF
26.# cat << EOF >> /mnt/etc/rc.conf
> zfs_enable=YES

27.# :> /mnt/etc/fstab
28.# zfs umount -a
29. # zfs set mountpoint=legacy sys/ROOT/default
30.# zfs set mountpoint=/home local/home
31.# zfs set mountpoint=/root local/root
32.# reboot
```

Listing 6. *Disk layout after Road Warrior Laptop install process*

```
# gpart show
=>      34  8388541  ada0  GPT  (4.0G)
        34      256     1  freebsd-boot  (128k)
       290  2097152     2  freebsd-zfs  (1.0G)
   2097442  6291133     3  freebsd-zfs  (3G)

# gpart list | grep label
   label: bootcode0
   label: sys0
   label: local0

# zpool status
  pool: local
 state: ONLINE
 scan: none requested
config:

        NAME              STATE     READ WRITE CKSUM
        sys               ONLINE       0     0     0
          gpt/local0.eli  ONLINE       0     0     0

errors: No known data errors

  pool: sys
 state: ONLINE
 scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        sys         ONLINE       0     0     0
          gpt/sys0  ONLINE       0     0     0

errors: No known data errors
```

References

[1] *http://forums.freebsd.org/showthread.php?t=10334*
[2] *http://forums.freebsd.org/showthread.php?t=12082*
[3] *http://docs.oracle.com/cd/E19963-01/pdf/820-6565.pdf*
[4] *http://docs.oracle.com/cd/E19963-01/html/821-1462/beadm-1m.html*
[5] *http://anonsvn.h3q.com/projects/freebsd-patches/wiki/manageBE*
[6] *https://sourceforge.net/projects/beadm/*
[7] *https://github/vermaden/beadm/*

Perform Upgrade within a Jail

This concept is about creating new boot environment from the desired one, lets call it `jailed`, then start that new environment inside a FreeBSD Jail and perform upgrade there.

After you have finished all tasks related to this upgrade and you are satisfied with the achieved results, shutdown that Jail, set the boot environment into that just upgraded Jail called `jailed` and reboot into just upgraded system without any risks.

- Create new boot environment called `jailed`.
```
# beadm create -e default jailed
Created successfully
```
- Create `/usr/jails` directory. `# mkdir /usr/jails`
- Set mount point of new boot environment to `/usr/jails/jailed` dir. `# zfs set mountpoint=/usr/jails/jailed sys/ROOT/jailed`
- Enable FreeBSD Jails mechanism and the `jailed` Jail in `/etc/rc.conf` file. `# cat << EOF >> /etc/rc.conf`
```
> jail_enable=YES
> jail_list="jailed"
> jail_jailed_rootdir="/usr/jails/jailed"
> jail_jailed_hostname="jailed"
> jail_jailed_ip="10.20.30.40"
> jail_jailed_devfs_enable="YES"
> EOF
```
- Start the Jails mechanism.
```
# /etc/rc.d/jail start
Configuring jails:.
Starting jails: jailed.
```
- Check if the jailed Jail started.
```
# jls
JID  IP Address  Hostname              Path
 1  10.20.30.40  jailed                /usr/jails/jailed
```
- Login into the `jailed` Jail. `# jexec 1 tcsh`
- *PERFORM ACTUAL UPGRADE.*
- Stop the `jailed` Jail.
```
# /etc/rc.d/jail stop
Stopping jails: jailed.
```
- Disable Jails mechanism in `/etc/rc.conf` file. `# sed -i '' -E s/"^jail _ enable.*$"/"jail _ enable=NO"/g /etc/rc.conf`
- Activate just upgraded `jailed` boot environment.
```
# bootfs-beadm activate jailed
Activated successfully
```
- Restart the system into upgraded system. `# shutdown -r now`

Import Boot Environment from Other Machine

Lets assume, that you need to upgrade or do some major modification to some of your servers, you will then create new boot environment from the default one, move it to other 'free' machine, perform these tasks there and after everything is done, move the modified boot environment to the production without any risks. You may as well transport that environment into You laptop/workstation and upgrade it in a Jail like in step 6.2 of this guide.

- Create new environment on the *production* server.
```
# beadm create upgrade
Created successfully.
```
- Send the `upgrade` environment to *test* server. `# zfs send sys/ROOT/upgrade | ssh TEST zfs recv -u sys/ROOT/upgrade`
- Activate the `upgrade` environment on the *test* server.
```
# beadm activate upgrade
Activated successfully.
```
- Reboot into the `upgrade` environment on the test server. `# shutdown -r now`
- *PERFORM ACTUAL UPGRADE AFTER REBOOT.*
- Sent the `upgraded` upgrade environment onto *production* server. `# zfs send sys/ROOT/upgrade | ssh PRODUCTION zfs recv -u sys/ROOT/upgrade`
- Activate upgraded `upgrade` environment on the *production* server.
```
# beadm activate upgrade
Activated successfully.
```
- Reboot into the `upgrade` environment on the *production* server. `# shutdown -r now`

The last part of the HOWTO remains the same as Year ago ...

You can now add your users, services and packages as usual on any FreeBSD system, have fun ;)

SŁAWOMIR WOJCIECH WOJTCZAK (VERMADEN)

Vermaden is another busy sysadmin with interest towards UNIX and BSD systems, often forced to also work with Linux. KISS principle follower.

OWNED!

Quick & Easy Security and Compliance Through RedSphere's Secure Hosting Solutions

- Instantly Become PCI Compliant
- We Address Other Compliance and Industry Security Requirements
  - Penetration Testing Services
  - Source Code Security Review and Design
  - Custom Security Services and Solutions

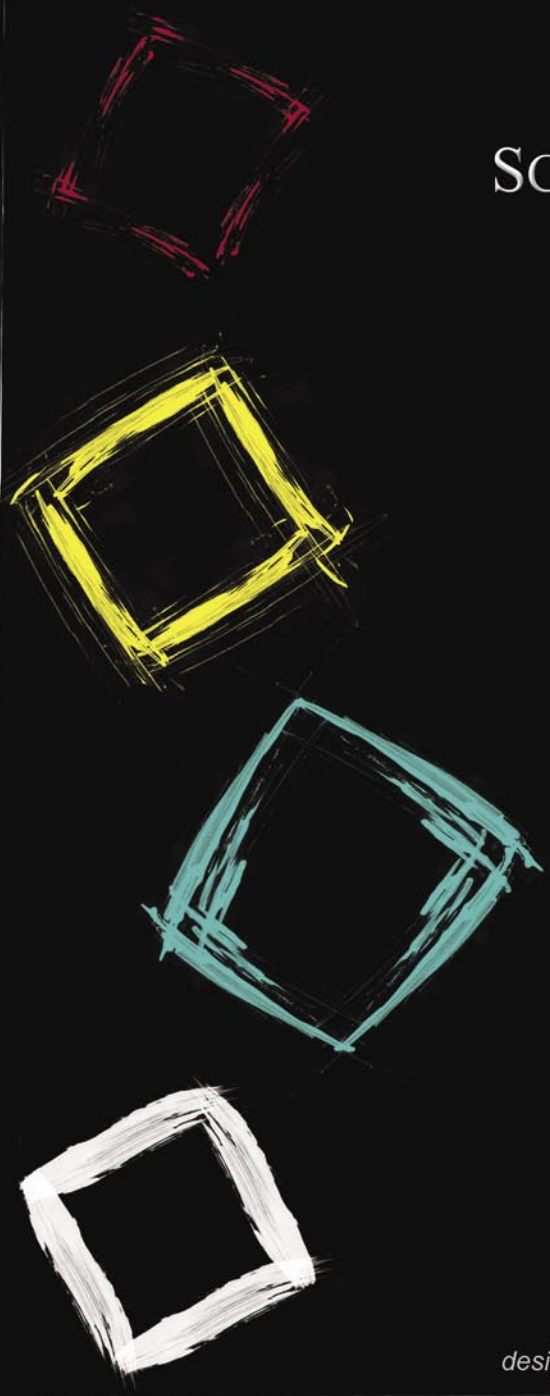
powered by FreeBSD®

REDSPHERE©™
Our Promise is Your Peace of Mind

RedSphere Global Security
Call now to speak to a representative
719.924.5266 sales@redsphereglobal.com

www.redsphereglobal.com

www.buildasearch.com



SCALABLE SITE SEARCH ENGINE

HOSTED SOLUTION

FAST SETUP

EASY TO USE WEB CONSOLE

ZERO CODING OPTION

AUTOMATED REPORTS

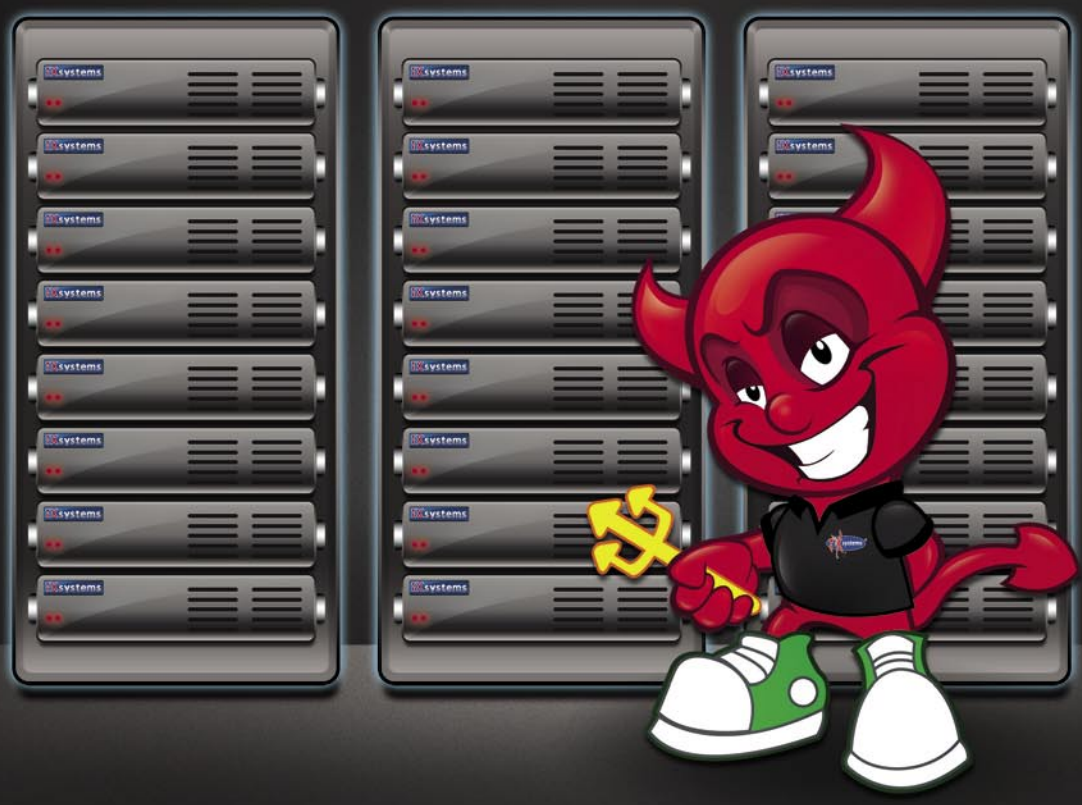
JSON AND XML RESULTS

POWERED BY FREEBSD

buildasearch

design by Emily Berry

What has your server vendor done for BSD lately? Probably, not much.



Work with a vendor that supports the operating system you love!

iX is the corporate sponsor of the PC-BSD® Project, a major corporate donor to the FreeBSD Foundation, and leads the FreeNAS™ development team -- all while employing some of the most brilliant minds in the FreeBSD® community. For BSD hardware and software expertise, look no further.

1-855-GREP-4-IX

http://www.iXsystems.com/community

iX systems®