Recent Security Enhancements in NetBSD

Elad Efrat <elad@NetBSD.org>

September 2006

Abstract

Over the years, NetBSD obtained the position of the BSD focusing on portability. While it is true that NetBSD offers an easily portable operating system, care is also given to other areas, such as security. This paper presents the NetBSD philosophy of security, design decisions, and currently offered security features. Finally, some of the current and future research will be revealed.

1. Introduction

Running on almost twenty different architectures, and easily portable to others, NetBSD gained its reputation as the most portable operating system on the planet. While that may indicate high quality code, the ever demanding networked world cares about more than just that. Over the past year, NetBSD evolved quite a bit in various areas; this paper, however, will focus on the aspect relating to security.

This paper was written and structured to present a full overview of the recent security enhancements in NetBSD in an easily readable and balanced form that will satisfy new, intermediate, and experienced users. References were sprinkled across the text to provide more information to those who want the gory details, while preserving the continuity.

Section 2 will present the bigger picture of security in NetBSD: how NetBSD perceives security, the design decisions of NetBSD software in general and the security infrastructure and features more specifically. Section 3 will present a detailed overview of the recent enhancements in the security infrastructure and features of NetBSD including, where relevant, details about the design, implementation, and possible future development. Section 4 will present current security-related research and development in NetBSD, and section 5 will discuss how the described enhancements work together to provide a more secure platform. Section 6 concludes the paper, and summarizes availability of discussed features.

2. The Tao of NetBSD Security

We are all familiar with the mantra that *security is a process, not a product*. When regarding software development, specifically operating systems, it should be part of the design, from the ground up. As the descendent of an operating system over 20 years old, NetBSD carries a security model designed and implemented with different threats in mind; the Internet was smaller, more naive, and less popular.

The following sections will provide background to the approach taken to enhance the security of the NetBSD operating system: the considerations, existing approaches, and case-studies.

2.1 Considerations

When approaching to enhance the security of NetBSD, two of the most important leading principles were maintaining compatibility and interoperability[1]. Presenting changes that would dramatically impact the user-base was out of question, and careful planning had to be done. In addition, any change to underlying back-ends had to be well thought-out so it maintains existing semantics without enforcing them during design stage.

2.2 Security Approaches

Operating system security is nothing new, and NetBSD is not the first to address the issue. In designing software – and security software in particular, it is mandatory to learn from the experience of previous work. Below are some common approaches to security and real-world case-studies.

2.2.1 Code Auditing

Code auditing addresses security issues by looking for programming glitches in the source code of the program, often with the assistant of automated tools[2]. Normally the work of vulnerability researchers, when done proactively by the programmers themselves, has the potential of locating and fixing bugs with security implications before anyone else finds and exploits them.

While some would argue that striving to produce bug-free code is the *one true way* of achieving security, this view is a fallacy for two main reasons. The first is that security issues are not always the result of programming errors; while code auditing tries to ensure no software bugs will be maliciously exploited because said bugs would simply not exist, it alone ignores other important aspects, such as configuration errors and user behavior policies.

The second reason is that it is not possible to write bug-free code[3]. Over the past decade, the awareness to writing secure code rose significantly; automated tools evolved, allowing easy pinpointing of software bugs; open-source software is available for the review of thousands – if not millions – of people; yet, we still see new security vulnerabilities on a daily basis. Some of those, ironically, are of the exact same type that affected us ten or twenty years ago[4].

2.2.2 Exploit Mitigation

The unorthodox approach of exploit mitigation addresses bugs from the opposite direction of code auditing: instead of looking for them in and removing them from software to make it more secure, it *adds bugs* to the exploit code to prevent it from working. While that may be over-simplified, the purpose of exploit mitigation technologies is to interfere with the inner-workings of the exploit, eliminating the – often unusual – conditions that make it work.

On one hand, some would claim that exploit mitigation discourages developers from writing secure code and vendors from quickly responding to security incidents: they

[1] Two other leading principles – not impacting the system performance and an easy user interface, will not be discussed in this paper.
[2] Coverity, for example, offered its services to various open-source projects, including NetBSD, for free. See http://scan.coverity.com
[3] http://www.cs.columbia.edu/~smb/papers/acm-predict.pdf
[4] http://www.cert.org/homeusers/buffer_overflow.html

know there's a *safety net* guarding them, and so they pay less attention to security when writing code, or taking their time coming up with fixes for security issues.

On the other hand, however, this is also where exploit mitigation technologies excel: they introduce the concept of preventing the successful exploitation of security vulnerabilities, even before a fix is available. Moreover, they prevent entire classes of bugs, and don't require constant updating.

2.2.3 Architectural Integration

So far, the previous two approaches assume the cause of a security breach is a bug in the code that is being exploited. The first approach tries to eliminate such bugs, and the second one tries to make it next to impossible to successfully exploit them. However, some environments require more than just that – for example, the ability to define detailed usage policies and associate them with entities on the system became a mandatory part of many security policies. In our context, we can relate that to the Unix permissions model; simply put, due to the coarse separation between a normal user and a superuser, it cannot be used to express many security policies as detailed as may be required.

That led to the research of various modern security models, of which most recognized ones are fine-grained[5] discretionary access controls (DACs) and mandatory access controls (MACs). To put things simple, DACs focus on the data owner's ability to specify who can use it and for what; MACs focus on a mandatory policy that affects everyone.

These systems allow an administrator – and where applicable, the users – to specify fine-grained policies; effectively, this means that a user or a program can be made to work with the minimal amount of privileges required for their operation (which, as implied above, cannot be done with the traditional Unix security model), resulting in damage containment in case of compromise or otherwise minimized impact from security vulnerabilities.

2.2.4 Layered security

To itself, layered security[6] is not a single approach. Where any of the previous three took a different route, the layered security approach suggests that maximized security can only be achieved by combining efforts on all fronts: code auditing is important, but does not come in place of useful exploit mitigation technologies; and architectural integration, of course, has little to do with any of them.

Although the above may sound obvious, it is not too often when you see an operating system that puts an emphasis on all three aspects; it will usually be the case that only one of the approaches is fully practiced. Following are some short case-studies that illustrate the importance of each approach by using real-world examples.

2.2.5 Case Studies

Shortly after splitting from NetBSD in 1995, OpenBSD became widely known for its unique – at the time – approach to security: proactive code auditing. Instead of

[5] I emphasize *fine-grained* because DACs already exist on Unix; however, as noted, they are too coarse.
[6] Also known as *Defense in Depth*.

retroactively responding to security issues, OpenBSD developers performed thorough code auditing sessions, *sweeping* for bugs. This act proved itself more than once, after vulnerabilities found in other operating systems were *already fixed*[7] in OpenBSD.

This, however, did not last too long. In 2002, winds of change blew through the OpenBSD mindset: the long standing fort of code auditing fell, adopting exploit mitigation technologies to its lap[8]. While the reasons behind the move were not published, some speculate that it was the release of an exploit allowing full system compromise of OpenBSD's default configuration[9] that finally proved that even a group of dedicated programmers cannot find all bugs; at least not first.

Said exploit mitigation technologies made their public debut around 1996, with the appearance of the Openwall[10] project, and later evolved dramatically by the PaX[11] project in 2000. Research done in both projects formed the basis of today's exploit mitigation technologies. Another commonality of the two was that they offered an implementation based on Linux – which only makes one wonder why it was OpenBSD that was the first to officially adopt these technologies.

Linux, however, took a different direction. First with the addition of POSIX.1e[12] capabilities in 1999, fine-grained discretionary access controls, later with SELinux[13], an implementation of mandatory access controls, and finally with the introduction of the Linux Security Modules framework[14], abstracting the implementation of both, Linux focused mainly on offering means for an administrator to define a detailed security policy, hoping to minimize the effect of a vulnerability.

Not lagging behind too much, though, exploit mitigation technologies also appeared in the official Linux kernel during 2004-2005; in fact, they also made an entrance to the official Windows world with Windows XP SP2[15], and Windows Vista is expected to include even more such technologies[16].

Simply put, all three major approaches have been practiced by widely used operating systems at one point or another. It is clear to see that although initially a single approach was chosen, eventually it was understood that layered security is the key to stronger defense of computer systems.

2.3 The NetBSD Perception of Security

Learning from others' experience, the approach taken by NetBSD employs three main principles:

- Simplicity. There is no point in providing a feature, whether it's a kernel subsystem or a userland tool, if it's not intuitive and easy to use. Furthermore, overly complex code is harder to audit, which may lead to additional bugs.

[7] http://www.openbsd.org/security.html#process
[8] http://www.monkey.org/openbsd/archive/misc/0207/msg01977.html
[9] http://www.securityfocus.com/news/493
[10] http://www.openwall.com
[11] http://pax.grsecurity.net
[12] http://wt.xpilot.org/publications/posix.1e/
[13] http://www.nsa.gov/selinux/papers/module/t1.html
[14] http://www.kroah.com/linux/talks/usenix_security_2002_lsm_paper/lsm.pdf
[15] http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx
[16] http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx

- Layered security. It is well understood that there is no single solution to security. NetBSD addresses security from a variety of angles, including code auditing, adequate and extensible security infrastructure, and exploit mitigation technologies.
- Sane defaults. Accepting that security may not be the highest priority for all users, NetBSD provides sane defaults to fit the common case. Detailed supplementary documentation helps enable and configure the various security features.

Using the above guidelines, a variety of security solutions were evaluated to address different threat models. With emphasis on implementing a solution that would fix a real problem and provide an intuitive and easy to use interface (when one is required), a variety of changes – ranging from tiny hooks, through additional kernel subsystems, to architectural modifications, NetBSD has made important first steps in improving its overall security.

3. Overview of Recent NetBSD Security Enhancements

3.1 Kernel Authorization

The introduction of kernel authorization, often referred to as kauth(9), in the NetBSD kernel has been one of the larger-scale changes ever done in NetBSD. The interface is modeled after an interface of the same name developed by Apple for Mac OS X[17], though unfortunately due to licensing issues it was impossible to make use of existing code, and so the NetBSD implementation was written completely from scratch.

Kernel authorization redefines the way credentials are handled by the kernel, and offers a simple and easy to use – yet powerful and extensible – kernel programming interface to enforce security policies. It is important to emphasize that kernel authorization does not provide any additional security by itself, but rather provides an interface on top of which security policies can be easily implemented. The strength of the security directly depends on the strength of the policy used.

The kernel authorization infrastructure is required for supporting fine-grained capabilities, ACLs, and pluggable security models among other things. It will allow NetBSD administrators and users to maintain the existing traditional Unix security model, offer capabilities to replace set-user-id and set-group-id programs, and allow third-party developers and appliance manufacturers to implement a custom security model to either replace or sit on-top of the existing one.

3.1.1 Related Work

Similar infrastructures are Linux's LSM (discussed earlier) and TrustedBSD's (now in FreeBSD) MAC framework[18]. Both have been in use for a couple of years, but like kernel authorization, are still very young to backup with real-world experiences.

3.1.2 Design

Apple did most of the design work for the kernel authorization infrastructure. A large part of the design is available online, and it’s merely the implementation that was

[17] http://developer.apple.com/technotes/tn2005/tn2127.html
[18] http://www.trustedbsd.org/trustedbsd-discex3.pdf

unavailable. Therefore, most of the design-related work in doing the native NetBSD port focused on completing the missing parts from the online documentation and taking care of compatibility issues.

Kernel authorization maps the privilege landscape of the kernel to *actions* grouped as *scopes*. For example, the *process scope* groups *actions* such as “can trace”, “can see”, and “can signal” – which are all operations on processes.

When a request for an operation is made, the *action* is passed to the *authorization wrapper* of the relevant scope, together with related context. The context is variable: it is different for each request. The authorization wrapper dispatches the request and the context to the *listeners* associated with the scope. Each listener can return a decision – either allow, deny, or defer (indicating the decision should be left to the other listeners) – and the authorization wrapper evaluates the responses from all listeners to decide whether to allow or deny the request.

In order for a request to be allowed, no listener may return a deny decision. If all listeners return a defer decision, the request is denied.

3.1.3 Implementation

The implementation of kernel authorization in NetBSD was done in several stages. First, the backend was written. This included the majority of the code that worked behind the scenes to implement the credential memory management and reference counting, locking, and scope and listener management. It was then tested to ensure all parts work as a black-boxes, allowing initial integration in the NetBSD code. Part of that work included merging the contents of the *ucred* and *pcred* structs into a single, opaque (as possible) type called *kauth_cred_t*.

The next step was a series of mechanical kernel programming interface changes. Credentials could no longer be allocated on the stack, and so a lot of code had to be modified to use the kauth(9) memory management routines. Additionally, code that directly referenced members of the *ucred* and *pcred* structures had to be modified to use the accessor and mutator routines provided by the kauth(9) interface. Existing interfaces such as suser(9) and groupmember(9) were deprecated in favor of calls to kernel authorization wrappers, and others such as sys_setgroups(2) and sys_getgroups(2) were modified to use the new interfaces.

The following step consisted of thorough testing – to ensure transparent integration and equivalent semantics – which uncovered some bugs with the kernel authorization code, most of them in the NFS portion of the kernel.

3.1.4 Future Development

While implementing the kernel authorization back-end and making the kernel dispatch its authorization requests to it was an important *ground preparation*, there is more work to be done before declaring this interface useful.

The first step in the integration of kernel authorization was mostly mechanical and transparent to users, intended to preserve existing semantics. The next logical step is to examine the kernel to ensure the interface abstracts the security model used in NetBSD.

Given its heritage, the NetBSD kernel is too tightly coupled with the Unix security model, and the concept of a single super-user with a user-id of zero is often hard-coded. For example, a lot of privileged operations check for an effective user-id of zero directly in the process' credentials structure, not making use of the suser(9) interface.

The next logical step will be to identify these locations, and properly replace these vague effective user-id checks with calls to the kernel authorization interface, describing the privilege required to complete the operation. The same applies to any authorization wrapper calls acting as placeholders, checking for super-user rights.

The above work will result in the complete abstraction of the security model used in the NetBSD kernel, allowing switching easy as a one-line change in the kernel configuration between the Unix security model, a finer-grained capabilities model, or a third-party security model possibly implemented using an LKM.

3.2 Veriexec

Veriexec is NetBSD's file-integrity subsystem, allowing the verification of a file's digital fingerprint before accessing it. Introduced in NetBSD by Brett Lymn in 2002[19] and later integrating work from Vexec of the Stephanie project[20] in 2005, Veriexec provides means to ensure real-time file integrity and monitoring combined with intrusion detection and prevention capabilities.

Initially self-contained, Veriexec's core – the interface for associating meta-data with files regardless of file-system support using in-kernel memory – was recently abstracted[21] to form the Fileassoc interface to satisfy similar needs from other features.

3.2.1 Related Work

Integrity checker implementations have been around for decades. Used for various purposes such as virus protection in DOS and file changes notifications in Unix, the concept itself is not new to the security industry. Dr. Fred Cohen's research was among the first to offer insight about using integrity checkers to protect from malicious software[22]. Tripwire[23], presented by Eugene Spafford and Gene Kim, allowed system administrators to be notified about corrupted or altered files in a timely fashion.

Yet, while there are numerous products for every computing environment, they all share a common set of flaws that prevents them from realizing their potential.

First, none of them integrates with the operating system deep enough to provide real-time protection: most are retroactive tools used to *notify* after changes were detected.

---

[19] http://mail-index.netbsd.org/tech-security/2002/10/30/0000.html
[20] http://ethernet.org/~brian/Stephanie/
[21] http://mail-index.netbsd.org/tech-kern/2006/06/08/0007.html
[22] http://vx.netlux.org/lib/afc03.html. Dr. Fred Cohen also introduced the concept of *integrity shells*, with which Veriexec is sharing some commonalities; no implementation was made available, however, and therefore it is impossible to tell whether the faults mentioned also apply to them.
[23] http://portal.acm.org/citation.cfm?id=191183

This approach does not address potential damage that be caused in the time-window between a file was altered and improperly used since and when the administrator receives notification of the matter and handles it. It also does not guarantee the integrity of the integrity checker itself: a successful compromise has the potential of remaining *under the radar*.

Furthermore, some implementations use weak algorithms[24] to calculate a file's checksum, or rely on a small data-set for checksum calculation. Other implementations rely on a file's attributes rather than data to evaluate integrity. The impact of the above is that a file can be modified in such ways that even if the integrity checker tries to evaluate it after the change, it will not be able to detect it. Whether it's by altering the file in a way to defeat the checksum algorithm, or modify areas of the file that the integrity checker is known to ignore, or even tamper with the file's attributes – these implementation flaws can all be bypassed by an attacker quite easily.

And last, they all leave out an important aspect in today's reality: the network. Our environments become more and more inter-connected; we access files from untrusted locations on a daily basis; some architectures rely on a networked environment for everyday operation: centralized storage, backup, and so on. Existing products may provide a certain level of local protection on a host, but leave an important – and interesting – question unanswered: how do you cope with the compromise of a remote resource?

While we cannot deal with all aspects of compromise of a remote resource we use, and it is certainly not our goal either, it is important to try and address the ones that can be solved by using an integrity checker integrated in the operating system.

3.2.2 Design

Veriexec was designed to be a file-system independent integrity subsystem protected from users, including root, by operating solely from the kernel. Recent attacks against various hashing algorithms once thought secure proven the need for interface flexibility – such that can be used both for easy addition of support for new hashing algorithms, as well as future work on digitally signed files.

Careful analysis of the bottlenecks for file access and other file-system semantics (such as rename and remove) resulted in generic hooks, to be called with the required context for decision-making and policy enforcement. At the time of writing, it is impossible to implement the Veriexec policy on top of kauth(9) due to lack of required scopes.

The design process also took into account various environments for Veriexec – from workstations, through servers and critical systems, to embedded task-oriented appliances. Strict levels with varying implications were introduced to support multiple uses, and were named semi-descriptively to hint for said uses: learning mode (level 0), intrusion detection system (IDS) mode (level 1), intrusion prevention system (IPS) mode (level 2), and lockdown mode (level 3).

[24] For example, CRC: https://www.kb.cert.org/vuls/id/25309

3.2.3 Implementation

The most recent version of Veriexec is implemented using the Fileassoc subsystem for management of meta-data and file association, greatly simplifying the Veriexec code, and a device for kernel-userland interaction.

Veriexec is implemented by hooking policy enforcement routines in various parts of the kernel, monitoring execution of normal executables as well as scripts, opening of regular files, and rename and remove operations.

When a file is opened or executed, the evaluation routine, is called with the context of the request (LWP, vnode, filename if any, and access flag indicating how the file was accessed) to make a decision whether the file can be accessed or not. The result is cached to speed-up further evaluations of the same file.

3.2.4 Future Development

During research work on Veriexec, Thor Lancelot Simon pointed out a potential attack[25]. Although Veriexec ensures integrity of files on local file-systems, where all access is done via the kernel, it cannot ensure integrity of files located on remote hosts, imported via NFS, for example.

While Veriexec could be told not to cache the evaluation of such files, the attack vector is when a process, or part of it, is paged-out and later paged in. Because the disk read is done by the VM system, and only of pieces (pages) of the program, Veriexec wasn't aware of it. If the remote host would be compromised, an attacker could write malicious data to the on-disk program, force a memory flush, which would later force a page-in, effectively injecting the malicious data into the address space of the running process on the Veriexec-protected host.

The remedy to this problem is in the form of per-page fingerprints. During fingerprint database generation, the administrator can add the *untrusted* flag to entries located on remote hosts. Veriexec will generate per-page fingerprints for them, and hook the VM system so that when a page-in occurs, the fingerprints of the relevant pages will be evaluated and compared to those calculated previously.

Another natural development for Veriexec would be to introduce support for digital signatures; that is discussed in subsection 4.2.

3.3 Exploit Mitigation

Exploit mitigation techniques are part of the layered security approach of NetBSD, complementing code auditing and more traditional security features, not intending on replacing them.

The purpose of exploit mitigation technologies is to interfere with the exploit code itself, preventing entire classes of exploits from working by short-circuiting common exploitation techniques. One popular example is making sure areas of the memory that are writable, such as the stack and the heap, are non-executable, and vice versa: areas that are executable, such as the where the program's code is, are not writable.

[25] http://mail-index.netbsd.org/tech-security/2002/11/01/0010.html

This prevents exploits that rely on injecting malicious code to a program's memory from working, because said code cannot be executed.

3.3.1 PaX MPROTECT

For a while NetBSD had support for non-executable mappings[26] on hardware platforms that allow it. However, experienced hackers have found a variety of ways to bypass them. Two of these are return-to-lib exploits[27] and trashing arguments to mprotect(2) to change the protection of memory[28].

The PaX MPROTECT[29] feature was developed to address the latter. It enforces a policy where memory that was once writable will not be able to later gain executable permission, and vice versa.

Naturally, this policy may break existing applications that make valid use of writable and executable memory, such as programs that load dynamic modules. For this reason, a tool is provided allowing marking executables as excluded from the PaX MPROTECT policy. It is also possible to revert the policy, applying it only to executables marked with an explicit enable flag.

While it is possible to modify programs that currently violate the PaX MPROTECT policy to continue working correctly without doing so, this would be an unfeasible effort with third-party applications.

3.3.2 SSP (Stack Smashing Protection) Compiler Extensions

Hiroaki Etoh developed SSP (also known as *ProPolice*) in IBM Research[30]. Its purpose is making exploitation of certain buffer overflows harder by placing random *canary values* right before the function return address on the stack, as well as reordering variables on the stack making it harder – if not impossible – to overflow stack buffers in order to overwrite integers or function pointers, preventing exploitation even without altering the return address.

First introduced in the OpenBSD 3.4 release, a similar functionality is now available in the stock gcc 4.1 compiler, recently integrated in NetBSD by Matthew Green.

3.3.3 Future Development

One of the planned features in this area for NetBSD is implementing PaX Address Space Layout Randomization[31]. Also developed by the PaX author, ASLR addresses exploitation via return-to-lib attacks[32] by randomizing the location in memory of shared libraries used by the application, thus making it a lot harder to correctly guess the location of library functions within the application address space.

---

[26] http://netbsd.org/Documentation/kernel/non-exec.html
[27] http://seclists.org/lists/bugtraq/1999/Mar/0004.html
[28] See thread http://seclists.org/dailydave/2004/q2/0045.html.
[29] http://pax.grsecurity.net/docs/mprotect.txt
[30] http://www.trl.ibm.com/projects/security/ssp/
[31] http://pax.grsecurity.net/docs/aslr.txt
[32] PaX ASLR addresses more than that; it also randomizes stack/heap base addresses for both userland and kernel threads.

As expected, hackers found ways to bypass ASLR. The two most commonly used attacks either combine an information leak bug leading to the disclosure of the location of libraries[33], or brute-force exploitation on respawning daemons in an attempt to guess the correct address in one of many attempts[34].

An ASLR implementation would not be complete without a solution to the latter technique. Such a solution, developed by Rafal Wojtczuk, is *Segvguard*[35], employing the basic concept of monitoring the rate of SIGSEGV signals sent to an application in a given time-frame, in an attempt to detect when a brute-force exploitation attack is taking place and prevent it by denying execution of the offending application.

A similar monitor will be introduced in NetBSD once ASLR is implemented.

3.4 Misc. Features

3.4.1 Information Filtering

One of the most common requirements from multi-user systems (such as public shell providers) is that users will not be able to tell what other users are doing – such as running programs, active network connections, login/logout times, etc.

NetBSD implements the above using the kernel authorization interface, and presents the administrator with a single knob that can be either enabled or disabled. When enabled, the authorization wrappers will match credentials of the two objects (the *looker* and the *lookee*) and return the decision.

This abstraction makes it easier to change the behavior of this feature in the future.

3.4.2 Strong Digital Checksum Support

Support for SHA2 checksums has been available in the NetBSD kernel for a while, mainly for the use of the IPSec network stack. Userland, however, was largely neglected. Tools such as cksum(1) and mtree(8) were able to make use only of hashes that were proven weak[36]. Given mtree(8) can be used to evaluate file-system integrity, this was rather dangerous.

The recent improvements to Veriexec, allowing it to support SHA2 hashes, amplified the need for userland support for SHA2 hashes, and were the trigger to adding SHA2 hash routines to libc, as well as support in cksum(1) and mtree(8).

3.4.3 Fileassoc

Fileassoc is one of the latest additions to the NetBSD kernel. It allows associating custom meta-data with files, independent of file-system support (such as extended attributes) using in-kernel memory. The interface is the result of research of other security features that stressed the need for an abstraction of code previously used exclusively by Veriexec.

3.4.3.1 Design

[33] http://artofhacking.com/files/phrack/phrack59/P59-0X09.TXT (mirror)
[34] http://artofhacking.com/files/phrack/phrack58/P58-0X04.TXT (mirror)
[35] Ibid.
[36] http://www.schneier.com/essay-074.html

The Fileassoc interface extends an already-existing design used by Veriexec. The requirements for the design were performance – so that using it in performance-critical code would not cause a notable impact on system performance – and ease of use. The interface was extended, allowing more than one *hook* to add its own file meta-data.

Designed with simplicity in mind, the interface allows multiple subsystems to hook private data on a per-file and/or per-device basis.

3.4.3.2 Implementation

To achieve the desired goal of near-zero performance impact of entry lookup, the Fileassoc subsystem makes use of hash tables and linked-lists to resolve collisions. The interface operates on *struct mount* * and *struct vnode* * to identify file-system mounts and files, respectively. While the internal implementation identifies a file as a pair of *struct mount* * and a file-id – the contents of *va_fileid* after a successful VOP_GETATTR() call – this is planned to change in the near future (see subsection 3.4.3.3).

In the current implementation, Fileassoc allows four hooks (which can be modified with a kernel option) to add private data to each file. This is transparent to the users of the interface, allowing changing in the future, if such is required.

3.4.3.3 Future Development

As previously mentioned, Fileassoc still relies on the *va_fileid* field as the unique identifier for files. This is an internal implementation detail, and expected to replaced in the future with file-handles by using calls to the file-system specific *vptofh()* routines.

3.4.4 Password Policy

Administrators often need to enforce a password policy on the system – either a system-global policy, per-application policy, or even a network-global policy. To address that issue, the password policy, or *pw_policy(3)*, interface was developed.

With flexibility and simplicity in mind, the *pw_policy(3)* interface was designed to allow an administrator to specify password policies via a collection of keywords, and applying them to named entities.

The interface is part of *libutil* and is small enough to be used from within any existing application. It was designed in a modular way, allowing future support for more keywords and evaluation routines.

4. Current NetBSD Security Research and Development

Discussed so far are solutions already implemented and available in NetBSD. Below you will find the current goals of the security research done in NetBSD, some of which are planned to be introduced as soon as the NetBSD 5.0 release.

4.1 Deprecating The Kernel Virtual Memory Interface kmem(4)

The kmem(4) device allows raw reading of kernel memory. It was introduced to allow programs that needed information from the kernel a way to extract it by reading the symbol list from the live kernel's on-disk image and seeking to it.

Several issues were raised regarding this device[37], and with 4.4BSD a new interface meant to replace kmem(4) was introduced, named *sysctl*. Sysctl allowed structured and controlled access to kernel information via syscalls carrying a *management information base* (MIB). The kernel held a tree-like hierarchy of information it can provide, and the MIB described what information is looked up.

From a security point of view, the kmem(4) device allows malicious processes running with *kmem* or *root* privileges to directly read or write kernel memory[38]. The attack vector here is widely abused[39] [40] mainly to introduce stealth rootkits into compromised systems.

Currently, NetBSD is doing loose usage of the kmem(4) interface, using it for more than a few userland utilities. There is an on-going effort to gradually convert programs using kmem(4) to sysctl with proper kernel support, allowing us to deprecate daily use of kmem(4) and maintain the interface for debugging needs only, if required.

4.2 Digitally Signed Files

At the moment, the Veriexec subsystem provides integrity based entirely on data. While it is strong enough to maintain file-system integrity on servers and critical systems, it lacks two important features: ability to securely modify the *baseline* during runtime, and ability to associate an identity with a file-system object.

Securely modifying the baseline during runtime is forbidden, even for the super-user, for security reasons: a possible scenario is that the host can be fully compromised and trojanned by an attacker; preventing the super-user from modifying critical programs can prevent that.

Associating a digital signature with a file-system object, regardless of implementation, could solve the above two by allowing an administrator to specify *trusted entities*. These could run any programs – as long as they are signed by them. That would mean that introducing a new program on the system required digital signing by a trusted entity, rather than a super-user adding its digital checksum to a database and rebooting.

It is planned to extend the Veriexec subsystem with this capability, in either one of two possible directions for the implementation; either delegating the digital signature processing to a user-space daemon, or making use of the BSD-licensed BPG inside the kernel.

4.3 Access Control Lists

Perhaps one of the longest remaining Unix relics in NetBSD is the file-system security model. Proven weak over time, modern operating systems implemented file-system access control lists, or *ACLs*.

---

[37] "The Design and Implementation of the 4.4BSD OS", pages 509-510.

[38] The use of raw access to bypass a security guard isn't limited to kernel memory: on-disk inodes could be modified using raw disk access, for example.

[39] http://artofhacking.com/files/phrack/phrack58/P58-0X07.TXT (mirror)

[40] "Rootkits: Subverting the Windows Kernel", chapter 7.

An ACL allows finer-grained file access, extending the *owner-group-other* scheme currently used.

There are two main issues when approaching file-system ACLs. The first is where to store them, and how to associate a potentially variable sized data-structure with a file. The second is what ACL model to use, which may dictate interoperability with other operating systems.

For the former, NetBSD provides both the UFS2 file-system[41], where extended attributes were introduced especially to address this issue, as well as the Fileassoc kernel interface, allowing file-system independent association of meta-data, after such data has been loaded via a driver.

Given recent standardization in ACL structure between Windows NT, Mac OS X, and NFSv4, it was decided to go with the same model for the latter, allowing NetBSD to properly operate in a heterogeneous environment.

4.4 Capabilities

Part of Unix's long-standing weaknesses is the use of set-id programs to elevate privileges of a normal user, either temporarily or permanently, required to complete an operation restricted to the super-user – for example, open a raw socket, bind to a reserved port, and so on.

The above lead to the absurdity that bugs in often trivial and non-critical programs could result in privilege abuse or even full system compromise.

Introducing capabilities, implemented as a set of kernel authorization listeners, will replace the role of the set-id bit in today's systems. Providing a fine-grained privilege model, each program will run with the minimal set of capabilities required for its operation. Furthermore, associating capabilities with users will allow us to define *user roles*, dividing the work-load of the super-user – possibly eliminating it entirely!

While a design for NetBSD capabilities hasn’t been laid out yet, it is expected that support for capabilities will be provided on the file-system layer, allowing the association of capabilities with a program using extended attributes, as well as an API a la OpenSolaris ppriv(3) for dropping unneeded capabilities during runtime, and a mechanism for associating capabilities with users on the system.

5. Component Interaction

So far the focus was on introducing the new infrastructure and features in NetBSD, as well as some on-going development. However, no emphasis was put on the interaction between the various components, and how they all cooperate and contribute to NetBSD's layered security model.

Throughout this section we'll examine the role of each feature in the layered security model.

[41] http://www.usenix.org/events/bsdcon03/tech/mckusick.html

5.1 Attack Vectors

Attacks can be conducted on various parts of the system, most commonly exploiting bugs in services (remote and local), misconfigurations, general program misuse, and user actions monitoring. Furthermore, post-compromise attacks include implanting trojan horses, backdoors, and rootkits.

Being a multipurpose operating system, NetBSD's security was designed to also be flexible and without a single point of failure: acknowledging different needs in different environments, the various security features are fully customizable, and the system is configured with sane defaults to ease administration.

5.2 Layer One: Exploit Mitigation and Privacy

In attempt to render an exploitation attempt itself as useless, the exploit mitigation features in NetBSD provide the first layer of security. The *curtain* hooks help protect the privacy of users in a multi-user environment, minimizing the potential of pre-attack information gathering and reconnaissance.

5.3 Layer Two: Capabilities

As discussed in subsection 4.4, capabilities are planned to replace the set-id bit. This effectively reduces the amount of privilege each program is running with. Successful exploitation of programs that today could result in pivoting[42] or super-user account compromise will result in a less critical privilege elevation in the worst case, limiting the impact of vulnerabilities on the overall security of the host.

5.4 Layer Three: Signed Files

Mentioned in subsection 4.2, signed files are the natural evolution of Veriexec, basically associating a signing entity with a file in addition to its digital fingerprint. The immediate benefit is obviously in introducing trust in networked environments, where files can be safely exchanged without fear of attacks such as man-in-the-middle[43].

Accessing files – in particular, running programs – that are signed by "trusted" entities in the default configuration could help reduce the possibility of running manipulated binaries even in face of attacks on the digital checksum algorithm. Doing so in the event of a compromise, combined with Veriexec's *lockdown mode*, will allow real-time investigation and remedy.

5.5 Layer Four: File-System Integrity

Interesting uses for Veriexec (presented in subsection 3.2) are its IDS and IPS modes. With functionality somewhat resembling a *fly-trap*, Veriexec in IDS mode can be used to silently monitor operations on critical system files (services, configuration files) in real-time, preventing any access to them once changed. This can make post-mortem analysis an easier task. IPS mode can be used to prevent access to these files altogether and generate proper log-files to help identify the source of the attack.

These two modes of operation can ensure file-system integrity even in the face of a super-user compromise, making it easier for an administrator to handle an attack

[42] Transition from one user to another.
[43] Assuming, of course, that the kernel itself cannot be manipulated.

without fear of trojanned, backdoored, or otherwise modified (via configuration files) services.

5.6 Layer Five: Protected Kernel Memory

Aimed at preserving kernel memory integrity, the work-in-progress for deprecating kmem(4) usage should result in the ability to remove the interface altogether[44], preventing the possibility of kernel memory manipulation by a malicious superuser on a compromised host. The benefit is obvious: no sophisticated rootkits or kernel-level backdoors can be implemented[45].

6. Conclusion

Throughout this paper I've outlined the recent enhancements in NetBSD security in terms of infrastructure and features, and how they conform to NetBSD's perception of security. Finally, I've exposed some on-going research and development, and showed how it all works together to create a more secure platform

While it is true that a lot of work is still ahead of us, this paper exposed the lot of work that is behind us. Over the past year NetBSD improved a lot on the security front, and it is expected that these efforts will pay off – if not already – within the next major release.

6.1 Availability

NetBSD 4.0 will include kernel authorization[46], PaX MPROTECT[47], GCC 4.1 with ProPolice, the information filtering hooks[48], fileassoc(9) [49], and pw_policy(3) [50].

Complete abstraction of the security model using kernel authorization is being considered for NetBSD 5.0, as well as PaX ASLR and a SegvGuard, Veriexec support for per-page fingerprints and digital signatures, file-system ACLs, and capabilities.

7. Credits

Thanks to the folks who reviewed this paper, either in part or in whole, helping improve its accuracy, readability, and quality. Jason V. Miller, Brian Mitchell and the guys at ISS, the PaX author, and Sean Trifero, Johnny Zackrisson, and Christos Zoulas.

Thanks to Brett Lymn, the PaX author, Bill Studenmund, YAMAMOTO Takashi, Matt Thomas, Jason R. Thorpe, and Christos Zoulas for helping with implementing the features discussed in this paper.

[44] From most systems. X would still require it without the use of an aperture driver.
[45] Unless, of course, a kernel vulnerability is successfully exploited.
[46] http://netbsd.gw.com/cgi-bin/man-cgi?kauth++NetBSD-current
[47] http://netbsd.gw.com/cgi-bin/man-cgi?paxctl++NetBSD-current
[48] See the *security.curtain* knob.
[49] http://netbsd.gw.com/cgi-bin/man-cgi?fileassoc++NetBSD-current
[50] http://netbsd.gw.com/cgi-bin/man-cgi?pw_policy++NetBSD-current. No programs were made aware of the interface yet, though.