Integration of Security Measures and Techniques in an Operating System

(considering OpenBSD as an example)

Igor Boehm

Institute for Information Processing and Microprocessor Technology
Johannes Kepler University Linz, Austria

Outline

Outline

The Basic Problem Being Studied

- The Clever Attacker:
  - ...finds a bug
  - ...knows how to craft an exploit
  - ...the exploit grants the attacker an advantage
  - ...the exploit is likely to work on many systems because of the `strict regularity` of the system environment
- **Is there a way to solve this problem?**

Outline

Simplified Solution for the Basic Problem

1. Make the system environment much more hostile towards exploitation.
2. Do not break behaviours programs depend on.
3. Try to change everything else which makes an exploit author cry.
4. Be careful about the performance hit.
5. Do not break any standards *(e.g. POSIX)*!
6. There should be no impact on well behaving processes!

Simplified Solution for the Basic Problem

1. Make the system environment much more hostile towards exploitation.
2. Do not break behaviours programs depend on.
3. Try to change everything else which makes an exploit author cry.
4. Be careful about the performance hit.
5. Do not break any standards *(e.g. POSIX)*!
6. There should be no impact on well behaving processes!

Simplified Solution for the Basic Problem

1. Make the system environment much more hostile towards exploitation.
2. Do not break behaviours programs depend on.
3. Try to change everything else which makes an exploit author cry.
4. Be careful about the performance hit.
5. Do not break any standards *(e.g. POSIX)*!
6. There should be no impact on well behaving processes!

Simplified Solution for the Basic Problem

1. Make the system environment much more hostile towards exploitation.
2. Do not break behaviours programs depend on.
3. Try to change everything else which makes an exploit author cry.
4. Be careful about the performance hit.
5. Do not break any standards *(e.g. POSIX)*!
6. There should be no impact on well behaving processes!

Simplified Solution for the Basic Problem

1. Make the system environment much more hostile towards exploitation.
2. Do not break behaviours programs depend on.
3. Try to change everything else which makes an exploit author cry.
4. Be careful about the performance hit.
5. Do not break any standards *(e.g. POSIX)*!
6. There should be no impact on well behaving processes!

Simplified Solution for the Basic Problem

1. Make the system environment much more hostile towards exploitation.
2. Do not break behaviours programs depend on.
3. Try to change everything else which makes an exploit author cry.
4. Be careful about the performance hit.
5. Do not break any standards *(e.g. POSIX)*!
6. There should be no impact on well behaving processes!

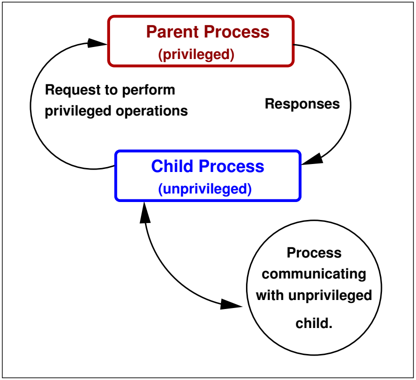
Outline

Limiting Risk with Privilege Separation

- Various daemon and system processes run with extra privileges.
- Those privileges are needed throughout the life-cycle of such processes for various tasks like:
  - allocation of a *socket*
  - reading and writing to and from certain files
  - adjusting the system time
  - ...
- The `goal` is to limit the risk of those extra privileges being compromised in the event of an attack.
- A way to solve this problem is to use `privilege separation`.

The Concept of Privilege Separation

- Set up two processes.
- One process is solely responsible for performing all privileged operations, `and it does absolutely nothing else`!
- The second process is responsible for performing the remainder of the program's work.



Privilege Separation Example

Privilege Separation Implemented in OpenNTPD (*Network Time Protocol Daemon*)

1. Initialisation Phase:
   - Setup a Unix domain socket pair.
   - Fork child process.
2. Privileged Parent *(ntpd)* - `Small Proces`:
   - Keep extra privileges.
   - Only perform little jobs that require privileges:
     - Correct the current system time by some offset.
     - Resolve hostnames.
3. Unprivileged Child *(ntp engine)* - `Large Process`:
   - Drop extra privileges in the child process.
   - Perform most tasks in the unprivileged child process:
     - Filter replies to increase accuracy.
     - Send queries to all peers.
     - Collapse the offsets learned from each peer into a single median offset.

Privilege Separation Example

Privilege Separation Implemented in OpenNTPD (*Network Time Protocol Daemon*)

1. Initialisation Phase:
   - Setup a Unix domain socket pair.
   - Fork child process.
2. Privileged Parent *(ntpd)* - `Small Proces`:
   - Keep extra privileges.
   - Only perform little jobs that require privileges:
     - Correct the current system time by some offset.
     - Resolve hostnames.
3. Unprivileged Child *(ntp engine)* - `Large Process`:
   - Drop extra privileges in the child process.
   - Perform most tasks in the unprivileged child process:
     - Filter replies to increase accuracy.
     - Send queries to all peers.
     - Collapse the offsets learned from each peer into a single median offset.

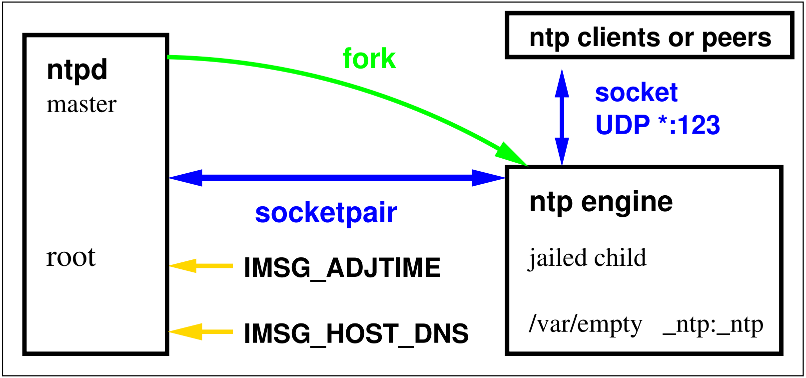
Privilege Separation Example

Privilege Separation Implemented in OpenNTPD (*Network Time Protocol Daemon*)

1. Initialisation Phase:
   - Setup a Unix domain socket pair.
   - Fork child process.
2. Privileged Parent *(ntpd)* - `Small Proces`:
   - Keep extra privileges.
   - Only perform little jobs that require privileges:
     - Correct the current system time by some offset.
     - Resolve hostnames.
3. Unprivileged Child *(ntp engine)* - `Large Process`:
   - Drop extra privileges in the child process.
   - Perform most tasks in the unprivileged child process:
     - Filter replies to increase accuracy.
     - Send queries to all peers.
     - Collapse the offsets learned from each peer into a single median offset.

Privilege Separation Example: OpenNTPD

Privilege Separation Implemented in OpenNTPD (*Network Time Protocol Daemon*)



Outline

1. Motivation
   - The Basic Problem Being Studied
   - Preliminary Solution Ideas and Goals

2. Security Solutions and Techniques
   - Secure Software Design Techniques
   - Memory Protection Techniques
   - Relevance of Random Numbers for Security

W^X - The Basics.

- Looking at the Operating Systems address space reveals that there is memory which is both `writeable` and `executable` (permissions = W | X) where it does not need to be!
- Because of this memory permission mess, many bugs are exploitable!
- This permission problem can best be solved by a generic policy for the whole address space with the following goals:
  - Each page may either be `writeable` or `executable`, but not both unless the application requests it.
  - Purify page permissions so that each page only has the minimum amount of permissions which are necessary!

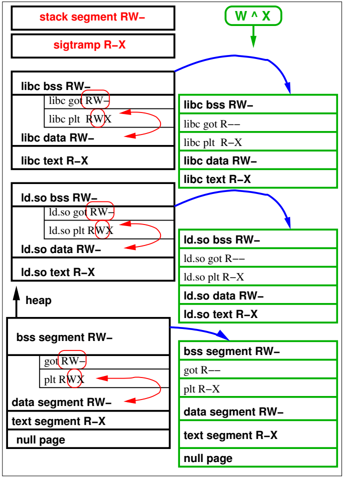
W^X - The Mechanism.

- The mechanism for an implementation of W^X depends on the MMU *(Memory Management Unit)* Architecture:
  - A `per page X bit` is supported by: *sparc, sparc64, alpha, amd64, ia64 and hppa*.
  - The *i386* architecture has a `code segment limit` where execution above a certain "line" does not work.
  - A `per segment X bit` is present for the *powerpc*.
- In order to support W^X a few process address space changes need to be done *(the amount of changes depends on the MMU)*.
- We are going to look at architectures which support the `per page X bit` and describe how the process address space has to be rearranged *(for architectures which lack the per page X bit, further information about how W^X is implemented can be found in the paper at http://www.bytelabs.org/papers.html)*.

W^X in Effect.

Example of Dynamic Library Mapping.

- Note that "data" segments are supposed to be only RW but contain objects which are RWX.
- Some objects are writeable when they do not need to be.
- Make a few things non writeable and give some objects their own pages in order to achieve W^X.
- Object descriptions:
  - .got: Global Offset Table
  - .plt: Procedure Linkage Table
  - .bss: Uninitialised data
  - .data: Initialised data
  - .text: Text or executable instructions



SSP - Stack Smashing Protector.

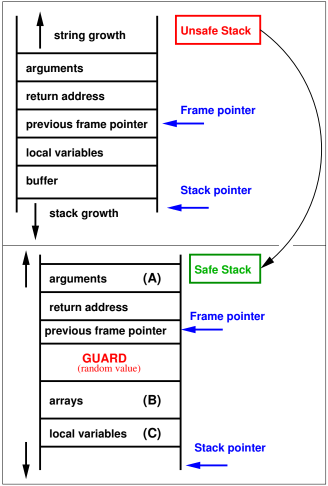
Improving the state of the art in buffer overflow detection.

- The `stack smashing protector` is a GCC *(Gnu Compiler Collection)* extension for protecting applications from stack-smashing attacks.
- Protects applications written in C by automatically inserting protection code for each function into an application at compilation time.
- Protection is realized by:
  - Buffer Overflow Detection:
    - Function Prologue stores a random value on the stack.
    - Function Epilogue aborts if value has changed.
  - Variable reordering feature to avoid the corruption of pointers.

SSP - Stack Smashing Protector.

A typical Stack Frame after a Function is called.

- Random Guard value is inserted by function prologue
- . . . and checked by function epilogue
- `Reordering` of `arrays` and `local variables` in order to avoid corruption of pointers.
- There is nothing which breaks as a result of this!
- It benefits security by finding bugs and making them unexploitable at a very low cost.



Outline

Random Numbers.

A Very General Overview.

- Generation of `randomness` with `deterministic` computers is very hard!
- Perfect randomness characterized by the `uniform distribution` is very hard to produce - instead `pseudo-random` generators are being used.
- `Pseudo-random` number generators have the goal that their output is computationally indistinguishable from the `uniform distribution`, while their execution must be feasible.
- Good random number generators depend on good sources of randomness which are usually chosen according to the following requirements:
  - they must be non-deterministic
  - they must be hard for an outside observer to measure

Random Numbers.

Gathering Entropy and Environmental Noise.

- The term `strong source of randomness` represents a generator whose output is not really random, but depends on so many entropy providing physical processes that an attacker can not practically predict its output.
- Examples of sources of randomness:
  - inter-keyboard timings
  - inter-interrupt timings
  - finishing time of disk requests
  - finishing time of net input
  - ...
- The measured values from these sources of randomness are added to an `entropy pool` by a mixing function in order to increase the pool's randomness.

Random Numbers.

Usage of Random Numbers.

- The 32-Bit sequence number field in the TCP header, a value which starts with a randomly generated arbitrary integer which then increments sequentially, is a place where a very fast and good random number generator is needed.
- The initialisation of volatile encryption keys requires a random number generator with a `strong source of randomness`.
- Since address space allocations and mappings are fairly predictable, randomization of address space is introduced and it heavily relies on a fast random number generator. This means that each time a program gets executed, it will show different address space behaviour and minimize the risk of an exploit which depends on the predictability of address space allocations.
- The `Guard` value which has been introduced in the Stack Smashing Protector also relies on a good and fast random number generator.
- Swap file encryption as a solution to prevent confidential data from remaining on a backing store relies on a fast random number generator.
- ...

Summary

- Security is like an arms race because the best attackers will continue to search for flaws.
- It is high time for defensive technologies which do not break any well behaving processes and have a low or non-existant performance hit.
- A good combination and integration of such defensive technologies and a proactive security approach, makes a system really secure.
- Security must be integrated into an Operating Systems design and not sold as an add on in order to be effective!

Thanks for your attention!