Software Security

Prof. Dr. Jean-Pierre Seifert
jpseifert@sec.t-labs.tu-berlin.de
http://www.sec.t-labs.tu-berlin.de/
Defenses against Memory Corruption
Preventing Buffer Overflows

- Use safe programming languages, e.g., Java
  - Legacy C code? Native-code library implementations?
- Black-box testing with long strings
- Mark stack as non-executable
- Randomize memory layout or encrypt return address on stack by XORing with random string
  - Attacker won’t know what address to use in his string
- Run-time checking of array and buffer bounds
  - StackGuard, libsafe, many other tools
- Static analysis of source code to find overflows
Reading

- Dhurjati, Adve. “Backwards-compatible array bounds checking for C with very low overhead” (ICSE 2006).
Run-Time Checking: StackGuard

- Embed “canaries” (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary

- Choose random canary string on program start
  - Attacker can’t guess what the value of canary will be

- Terminator canary: “\0”, newline, linefeed, EOF
  - String functions like strcpy won’t copy beyond “\0”
StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- StackGuard can be defeated
  - A single memory copy where the attacker controls both the source and the destination is sufficient
Defeating StackGuard

- Suppose program contains `strcpy(dst,buf)` where attacker controls both dst and buf
  - Example: dst is a local pointer variable

```
buf dst canary sfp RET
```

Return execution to this address

```
BadPointer, attack code &RET canary sfp RET
```

Overwrite destination of strcpy with RET position

strcpy will copy BadPointer here
ProPolice / SSP

[IBM, used in gcc 3.4.1; also MS compilers]

- Rerrange stack layout (requires compiler mod)

  - Args
  - Return address
  - Exception handler records
  - SFP
  - CANARY
  - Arrays
  - Local variables

  - No arrays or pointers
  - Ptrs, but no arrays

  String growth

  Stack growth

  Cannot overwrite any pointers by overflowing an array
What Can Still Be Overwritten?

- Other string buffers in the vulnerable function
- Exception handling records
- Any stack data in functions up the call stack
  - Example: call to a vulnerable member function passes as an argument `this` pointer to an object up the stack
  - Stack overflow can overwrite this object’s vtable pointer and make it point into an attacker-controlled area
  - When a virtual function is called (how?), control is transferred to attack code (why?)
  - Do canaries help in this case?
    - Hint: when is the integrity of the canary checked?
Litchfield’s Attack

- Microsoft Windows 2003 server implements several defenses against stack overflow
  - Random canary (with /GS option in the .NET compiler)
  - When canary is damaged, exception handler is called
  - Address of exception handler stored on stack above RET

- Litchfield’s attack (see paper)
  - Smashes the canary AND overwrites the pointer to the exception handler with the address of the attack code
    - Attack code must be on the heap and outside the module, or else Windows won’t execute the fake “handler”
  - Similar exploit used by CodeRed worm
Safe Exception Handling

- Exception handler record must be on the stack of the current thread (why?)
- Must point outside the stack (why?)
- Must point to a valid handler
  - Microsoft’s /SafeSEH linker option: header of the binary lists all valid handlers
- Exception handler records must form a linked list, terminating in FinalExceptionHandler
  - Windows Server 2008: SEH chain validation
  - Address of FinalExceptionHandler is randomized (why?)
When SafeSEH Is Incomplete

- If DEP is disabled, handler is allowed to be on any non-image page except stack
  - Put attack code on the heap, overwrite exception handler record on the stack to point to it

- If any module is linked without /SafeSEH, handler is allowed to be anywhere in this module
  - Overwrite exception handler record on the stack to point to a suitable place in the module
  - Used to exploit Microsoft DNS RPC vulnerability in Windows Server 2003
PointGuard

- Attack: overflow a function pointer so that it points to attack code

- Idea: **encrypt all pointers** while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflowed while in registers

- Attacker cannot predict the target program’s key
  - Even if pointer is overwritten, after XORing with key it will dereference to a “random” memory address
Normal Pointer Dereference

1. Fetch pointer value
2. Access data referenced by pointer

Memory

- Pointer 0x1234
- Data 0x1234

CPU

1. Fetch pointer value
2. Access attack code referenced by corrupted pointer

Memory

- Corrupted pointer 0x1234 0x1340
- Data 0x1234
- Attack code 0x1340

[Cowan]
PointGuard Dereference

1. Fetch pointer value
2. Access data referenced by pointer

Memory

CPU

Decrypt
0x1234

Encrypt 0x7239
pointer

Data

0x1234

CPU

Decrypt
0x9786

Decrypts to random value

1. Fetch pointer value
2. Access random address; segmentation fault and crash

Memory

Encrypt 0x7239

Attack code

0x1234
0x1340
0x9786

Corrupted pointer
0x7239
0x1340

Data

0x1234
0x1340

[Cowan]
PointGuard Issues

- Must be very fast
  - Pointer dereferences are very common

- Compiler issues
  - Must encrypt and decrypt only pointers
  - If compiler “spills” registers, unencrypted pointer values end up in memory and can be overwritten there

- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page

- PG’d code doesn’t mix well with normal code
  - What if PG’d code needs to pass a pointer to OS kernel?
Run-Time Checking: Libsafe

- Dynamically loaded library
- Intercepts calls to strcpy(dest, src)
  - Checks if there is sufficient space in current stack frame
    \[ |\text{frame-pointer} - \text{dest}| > \text{strlen}(\text{src}) \]
  - If yes, does strcpy; else terminates application
Limitations of Libsafe

- Protects frame pointer and return address from being overwritten by a stack overflow
- Does not prevent sensitive local variables below the buffer from being overwritten
- Does not prevent overflows on global and dynamically allocated buffers
**TIED / LibsafePlus**

- **TIED**: augments the executable with size information for global and automatic buffers
- **LibsafePlus**: intercepts calls to unsafe C library functions and performs more accurate and extensive bounds checking

[Avijit et al.]
Overall Approach

Executable compiled with -g option

TIED

Augmented executable

LibsafePlus.so

Preload

Run

Aborts if buffer overflow

Normal execution otherwise
TIED: The Binary Rewriter

- Extracts type information from the executable
  - Executable must be compiled with -g option
- Determines location and size for automatic and global character arrays
- Organizes the information as tables and puts it back into the binary as a loadable, read-only section
## Type Information Data Structure

**Type info header pointer**
- No. of global variables
- Ptr to global var table
- No. of functions
- Ptr to function table

**Global Variable Table**
<table>
<thead>
<tr>
<th>Starting address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Local Variable Table**
<table>
<thead>
<tr>
<th>Offset from frame pointer</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Function Table**
<table>
<thead>
<tr>
<th>Starting address</th>
<th>End address</th>
<th>No. of vars</th>
<th>Ptr to var table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Rewriting ELF Executables

- **Constraint:** the virtual addresses of existing code and data should not change
- Extend the executable towards lower virtual addresses by a multiple of page size
- Serialize, relocate, and dump type information as a new loadable section in the gap created
- Provide a pointer to the new section as a symbol in the dynamic symbol table
Before and After Rewriting

.dynstr is modified to hold the name of the symbolic pointer.

.hash is modified to hold the hash value of the symbol added to .dynsym.
Bounds Checking by LibsafePlus

- Intercept unsafe C library functions
  - strcpy, memcpy, gets ...
- Determine the size of destination buffer
- Determine the size of source string
- If destination buffer is large enough, perform the operation using actual C library function
- Terminate the program otherwise
Estimating Stack Buffer Size

- Preliminary check: is the buffer address greater than the current stack pointer?
- Locate the encapsulating stack frame by traversing the saved frame pointers
- Find the function that defines the buffer
- Search for the buffer in the local variable table corresponding to the function
  - This table has been added to the binary by TIED
- Return the loose Libsafe bound if buffer is not present in the local variable table
Where Was The Buffer Defined?

Case 1: \texttt{buf} may be local variable of function \texttt{f} or

Case 2: \texttt{buf} may be an argument to the function \texttt{g}

Use return address into \texttt{f} to locate the local variable table of \texttt{f}, search it for a matching entry.

If no match is found, repeat the step using return address into \texttt{g}.
Protecting Heap Variables

- LibsafePlus also provides protection for variables allocated by malloc family of functions
- Intercepts calls to malloc family of functions
- Records sizes and addresses of all dynamically allocated chunks in a red-black tree.
  - Used to find sizes of dynamically allocated buffers
- Insertion, deletion and searching in $O(\log(n))$
Estimating Heap Buffer Size

- Maintain the smallest starting address M returned by malloc family of functions
- Preliminary check: if the buffer is not on the stack, is its address greater than M?
- If yes, search in the red-black tree to get the size
- If buffer is neither on stack, nor on heap, search in the global variable table of the type information data structure
Limitations of TIED / LibsafePlus

- Does not handle overflows due to erroneous pointer arithmetic
- Imprecise bounds for automatic variable-sized arrays and alloca()’ed buffers
- Applications that mmap() to fixed addresses may not work
- Type information about buffers inside shared libraries is not available
  - Addressed in a later version
Runtime Bounds Checking

Referent object = buffer to which pointer points
- Actual size is available at runtime!

1. Modified pointer representation
   - Pointer keeps information about its referenced object
   - Incompatible with external code, libraries, etc. 😞

2. Special table maps pointers to referent objects
   - Check referent object on every dereference
   - What if a pointer is modified by external code? 😞

3. Keep track of address range of each object
   - For every pointer arithmetic operation, check that the result points to the same referent object
Pad each object by 1 byte
   - C permits a pointer to point to the byte right after an allocated memory object

Maintain a runtime tree of allocated objects

Backwards-compatible pointer representation

Replace all out-of-bounds addresses with special ILLEGAL value (if dereferenced, program crashes)

Problem: what if a pointer to an out-of-bounds address is used to compute an in-bounds address
   - Result: false alarm
Example of a False Alarm

```c
{ 
    char *p, *q, *r, *s;
    p = malloc(4);
    q = p+1;
    s = p+5;
    r = s-3;
}
```

S is set to ILLEGAL

Program will crash if r is ever dereferenced

Referent object (4 bytes)

Note: this code works even though it’s technically illegal in standard C
Ruwase-Lam

- Catch out-of-bounds pointers at runtime
  - Requires instrumentation of malloc() and a special runtime environment

- Instead of ILLEGAL, make each out-of-bounds pointer point to a special OOB object
  - Stores the original out-of-bounds value
  - Stores a pointer to the original referent object

- Pointer arithmetic on out-of-bounds pointers
  - Simply use the actual value stored in the OOB object

- If a pointer is dereferenced, check if it points to an actual object. If not, halt the program!
Example of an OOB Object

```c
{ char *p, *q, *r, *s; p = malloc(4); q = p+1; s = p+5; r = s-3; }
```

Value of `r` is in bounds

Note: this code works even though it’s technically illegal in standard C
Performance

- Checking the referent object table on every pointer arithmetic operation is very expensive
- Jones-Kelly: 5x-6x slowdown
  - Tree of allocated objects grows very big
- Ruwase-Lam: 11x-12x slowdown if enforcing bounds on all objects, up to 2x if only strings
- Unusable in production code!
Dhurjati-Adve

- Split memory into disjoint pools
  - Use aliasing information
  - Target pool for each pointer known at compile-time
  - Can check if allocation contains a single element (why does this help?)

- Separate tree of allocated objects for each pool
  - Smaller tree ⇒ much faster lookup; also caching

- Instead of returning a pointer to an OOB, return an address from the kernel address space
  - Separate table maps this address to the OOB
  - Don’t need checks on every dereference (why?)
OOB Pointers: Ruwase-Lam

```
p = malloc(10 * sizeof(int));
q = p + 20;

r = q - 15;
*r = ... ; //no bounds overflow
*q = ... ; // overflow
```

Check if q is out of bounds:
Runtime error

Check if r is out of bounds:
Check on every dereference

q = OOB(p+20,p)
Put OOB(p+20,p) into a map

r = p + 5
OOB Pointers: Dhurjati-Adve

q = 0xCCCCCCCC
Put (0xCCCCCCCC, OOB(p+20,p)) into a map

p = malloc(10 * sizeof(int));
q = p + 20;
r = q - 15;
*r = ... ; //no bounds overflow

r = p + 5

No software check necessary!

*q = ... ; // overflow

No software check necessary!
Runtime error

Average overhead: 12% on a set of benchmarks
Reading


Optional:
- PaX documentation (http://pax.grsecurity.net/docs/)
Problem: Lack of Diversity

- Buffer overflow and return-to-libc exploits need to know the (virtual) address to hijack control
  - Address of attack code in the buffer
  - Address of a standard kernel library routine
- Same address is used on many machines
  - Slammer infected 75,000 MS-SQL servers using same code on every machine
- Idea: introduce artificial diversity
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine
ASLR

- Address Space Layout Randomization
- Randomly choose base address of stack, heap, code segment
- Randomly pad stack frames and malloc() calls
- Randomize location of Global Offset Table
- Randomization can be done at compile- or link-time, or by rewriting existing binaries
  - Threat: attack repeatedly probes randomized binary
PaX

- Linux kernel patch
- Goal: prevent execution of arbitrary code in an existing process’s memory space
- Enable executable/non-executable memory pages
- Any section not marked as executable in ELF binary is non-executable by default
  - Stack, heap, anonymous memory regions
- Access control in mmap(), mprotect() prevents unsafe changes to protection state at runtime
- Randomize address space layout
Non-Executable Pages in PaX

- In older x86, pages cannot be directly marked as non-executable
- PaX marks each page as “non-present” or “supervisor level access”
  - This raises a page fault on every access
- Page fault handler determines if the fault occurred on a data access or instruction fetch
  - Instruction fetch: log and terminate process
  - Data access: unprotect temporarily and continue
mprotect() in PaX

- mprotect() is a Linux kernel routine for specifying desired protections for memory pages
- PaX modifies mprotect() to prevent:
  - Creation of executable anonymous memory mappings
  - Creation of executable and writable file mappings
  - Making executable, read-only file mapping writable
    - Except when relocating the binary
  - Conversion of non-executable mapping to executable
Access Control in PaX mprotect()

- In standard Linux kernel, each memory mapping is associated with permission bits
  - VM_WRITE, VM_EXEC, VM_MAYWRITE, VM_MAYEXEC
    - Stored in the vm_flags field of the vma kernel data structure
    - 16 possible write/execute states for each memory page

- PaX makes sure that the same page cannot be writable AND executable at the same time
  - Ensures that the page is in one of the 4 “good” states
    - VM_MAYWRITE, VM_MAYEXEC, VM_WRITE | VM_MAYWRITE, VM_EXEC | VM_MAYEXEC
  - Also need to ensure that attacker cannot make a region executable when mapping it using mmap()
PaX ASLR

- User address space consists of three areas
  - Executable, mapped, stack
- Base of each area shifted by a random “delta”
  - Executable: 16-bit random shift (on x86)
    - Program code, uninitialized data, initialized data
  - Mapped: 16-bit random shift
    - Heap, dynamic libraries, thread stacks, shared memory
    - Why are only 16 bits of randomness used?
  - Stack: 24-bit random shift
    - Main user stack
PaX RANDUSTACK

- Responsible for randomizing userspace stack
- Userspace stack is created by the kernel upon each execve() system call
  - Allocates appropriate number of pages
  - Maps pages to process’s virtual address space
    - Userspace stack is usually mapped at 0xBFFFFFFF, but PaX chooses a random base address
- In addition to base address, PaX randomizes the range of allocated memory
PaX RANDKSTACK

- Linux assigns two pages of kernel memory for each process to be used during the execution of system calls, interrupts, and exceptions.
- PaX randomizes each process’s kernel stack pointer before returning from kernel to userspace.
  - 5 bits of randomness.
- Each system call is randomized differently.
  - By contrast, user stack is randomized once when the user process is invoked for the first time.
PaX RANDMMAP

- Linux heap allocation: `do_mmap()` starts at the base of the process’s unmapped memory and looks for the first unallocated chunk which is large enough

- PaX: add a random `delta_mmap` to the base address before looking for new memory
  - 16 bits of randomness
**PaX RANDEXEC**

- Randomizes location of ELF binaries in memory
- Problem if the binary was created by a linker which assumed that it will be loaded at a fixed address and omitted relocation information
  - PaX maps the binary to its normal location, but makes it non-executable + creates an executable mirror copy at a random location
  - Access to the normal location produces a page fault
  - Page handler redirects to the mirror “if safe”
    - Looks for “signatures” of return-to-libc attacks and may result in false positives
Base-Address Randomization

- Only the base address is randomized
  - Layouts of stack and library table remain the same
  - Relative distances between memory objects are not changed by base address randomization
- To attack, it’s enough to guess the base shift
- A 16-bit value can be guessed by brute force
  - Try $2^{15}$ (on average) overflows with different values for addr of known library function – how long does it take?
    - Shacham et al. attacked Apache with return-to-libc
    - usleep() is used (why?)
  - If address is wrong, target will simply crash
ASLR in Windows

- Vista and Server 2008
- Stack randomization
  - Find $N^{th}$ hole of suitable size ($N$ is a 5-bit random value), then random word-aligned offset (9 bits of randomness)
- Heap randomization: 5 bits
  - Linear search for base + random 64K-aligned offset
- EXE randomization: 8 bits
  - Preferred base + random 64K-aligned offset
- DLL randomization: 8 bits
  - Random offset in DLL area; random loading order
Bypassing Windows ASLR

- Implementation uses randomness improperly, thus distribution of heap bases is biased
  - Ollie Whitehouse’s paper (Black Hat 2007)
  - Makes guessing a valid heap address easier
- When attacking browsers, may be able to insert arbitrary objects into the victim’s heap
  - Executable JavaScript code, plugins, Flash, Java applets, ActiveX and .NET controls...
- Heap spraying
  - Stuff heap with large objects and multiple copies of attack code (how does this work?)
Example: Java Heap Spraying

- JVM makes all of its allocated memory RWX: readable, writeable, executable (why?)
  - Yay! DEP now goes out the window...
- 100MB applet heap, randomized base in a predictable range
  - 0x20000000 through 0x25000000
- Use a Java applet to fill the heap with (almost) 100MB of NOP sleds + attack code
- Use your favorite memory exploit to transfer control to 0x25A00000 (why does this work?)

[See Sotirov & Dowd]
Information Leaks Break ASLR

- User-controlled .NET objects are not RWX
- But JIT compiler generates code in RWX memory
  - Can overwrite this code or “return” to it out of context
  - But ASLR hides location of generated code stubs...
  - Call `MethodHandle.GetFunctionPointer()` ... .NET itself will tell you where the generated code lives!

- ASLR is often defeated by information leaks
  - Pointer betrays an object’s location in memory
    - For example, a pointer to a static variable reveals DLL’s location... for all processes on the system! (why?)
  - Pointer to a frame object betrays the entire stack

[See Sotirov & Dowd]
.NET Address Space Spraying

- Webpage may embed .NET DLLs
  - No native code, only IL bytecode
  - Run in sandbox, thus no user warning (unlike ActiveX)
  - Mandatory base randomization when loaded
- Attack webpage include a large (>100MB) DLL

[See Sotirov & Dowd]
Dealing with Large Attack DLLs

- 100MB is a lot for the victim to download!

- Solution 1: binary padding
  - Specify a section with a very large VirtualSize and very small SizeOfRawData – will be 0-padded when mapped
  - On x86, equivalent to `add byte ptr [eax], al` - NOP sled!
    - Only works if EAX points to a valid, writeable address

- Solution 2: compression
  - gzip content encoding
    - Great compression ratio, since content is mostly NOPs
  - Browser will unzip on the fly
Spraying with Small DLLs

- Attack webpage includes many small DLL binaries
- Large chunk of address space will be sprayed with attack code

![Small DLL Mapping Diagram]

- Small DLLs placed contiguously
- All are lying on 64K boundaries

See Sotirov & Dowd
Turning Off ASLR Entirely

- Any DLL may “opt out” of ASLR
  - Choose your own ImageBase, unset IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag
- Unfortunately, ASLR is enforced on IL-only DLL
- How does the loader know a binary is IL-only?

```c
if( ( (pCORHeader->MajorRuntimeVersion > 2) ||
    (pCORHeader->MajorRuntimeVersion == 2 && pCORHeader->MinorRuntimeVersion >= 5) ) &&
    (pCORHeader->Flags & COMIMAGE_FLAGS_ILONLY) )
{
    pImageControlArea->pBinaryInfo->pHeaderInfo->bFlags |= PINFO_IL_ONLY_IMAGE;
    ...
}
```

Set version in the header to anything below 2.5 ASLR will be disabled for this binary!
Bypassing IL Protections

- Embedded .NET DLLs are expected to contain IL bytecode only - many protection features
  - Verified prior to JIT compilation and at runtime, DEP
  - Makes it difficult to write effective shellcode
- ... enabled by a single global variable
  - `mscorwks!s_eSecurityState` must be set to 0 or 2
  - Does `mscorwks` participate in ASLR? No!
- Similar: disable Java bytecode verification
  - JVM does not participate in ASLR, either
  - To disable runtime verification, traverse the stack and set NULL protection domain for current method

[Dowd & Sotirov, PacSec 2008]
Ideas for Better Randomization (1)

- 64-bit addresses
  - At least 40 bits available for randomization
    - Memory pages are usually between 4K and 4M in size
  - Brute-force attack on 40 bits is not feasible

- Does more frequent randomization help?
  - ASLR randomizes when a process is created
  - Alternative: re-randomize address space while brute-force attack is still in progress
    - E.g., re-randomize non-forking process after each crash (recall that unsuccessful guesses result in target’s crashing)
  - This does not help much (why?)
Ideas for Better Randomization (2)

- Randomly re-order entry points of library functions
  - Finding address of one function is no longer enough to compute addresses of other functions
    - What if attacker finds address of system()?

- ... at compile-time
  - Access to source, thus no virtual memory constraints; can use more randomness (any disadvantages?)

- ... or at run-time
  - How are library functions shared among processes?
  - How does normal code find library functions?
Comprehensive Randomization (1) [Bhatkar et al.]

- Function calls
  - Convert all functions to function pointers and store them in an array
  - Reorder functions within the binary
  - Allocation order of arguments is randomized for each function call

- Indirect access to all static variables
  - Accessed only via pointers stored in read-only memory
  - Addresses chosen randomly at execution start
Comprehensive Randomization (2)

- Locations of stack-allocated objects randomized continuously during execution
  - Separate shadow stack for arrays
  - Each array surrounded by inaccessible memory regions

- Insert random stack gap when a function is called
  - Can be done right before a function is called, or at the beginning of the called function (*what’s the difference?*

- Randomize heap-allocated objects
  - Intercepts malloc() calls and requests random amount of additional space

[Bhatkar et al.]
Comprehensive Randomization (3)

- Randomize base of stack at program start
- Shared DLLs *(see any immediate issues?)*
- Procedure Linkage Table/Global Offset Table
- setjmp/longjmp require special handling
  - Must keep track of context (e.g., shadow stack location)
Summary

- Randomness is a potential defense mechanism
- Many issues for proper implementation
- Serious limitations on 32-bit architecture
  - "Thus, on 32-bit systems, runtime randomization cannot provide more than 16-20 bits of entropy"
    – Shacham et al.
Reading

- Dhurjati, Adve. “Backwards-compatible array bounds checking for C with very low overhead” (ICSE 2006).
- PaX documentation (http://pax.grsecurity.net/docs/)
Thank you for your attention!

Questions?