# PaX - gcc plugins galore

PaX Team

H2HC 2013.10.05

Introduction
- PaX/grsecurity

GCC & plugins

Instrumentation 1

Instrumentation 2

# Overview

- Host Intrusion Prevention System
- Focus: exploitation of memory corruption bugs
- Threat model: arbitrary read-write memory access
- Bugs vs. Exploits vs. Exploit techniques
- Performance vs. Usability
- 2000-2013, linux 2.2-3.11

# Exploit Techniques & Defenses

- Execute new (injected) code (shellcode) → non-executable pages, runtime code generation control, ASLR
- Execute existing code out-of-(intended)-order (return-to-libc, ROP/JOP) → control flow integrity, ASLR
- Execute existing code in-(intended)-order (data-only attacks) → open question
- Increasing order of difficulty
- Decreasing amount of control

# Memory Corruption Bugs

- “Precursor” bugs included (memory disclosure, unintended reads, etc)
- Two generic goals:
  - Find them in the source
  - Catch them before they trigger
- Too many kinds to cover them with universal approaches
- see http://cwe.mitre.org/

# Why GCC Plugins?

- De facto compiler in the linux world
- Compiler is the bridge between source code and machine code
- Read(analysis)/Write(instrumentation) access to the internal representation of the program
- Access to all kinds of meta information for free (CFG, data flow, etc)
  - Idea: add special instrumentation during compilation to detect/prevent entire bug classes at runtime
- You'll learn C for real :)

- GCC = GNU C Compiler, GNU Compiler Collection, FSF's flagship project
- Languages: C, C++, Objective-C, Objective-C++, Go, Ada, Java, Fortran, GIMPLE
- GCC itself is written in C (and since 4.7 more and more C++)
- C dialects: C90 (c90, gnu90), C99 (c99, gnu99), C11 (c11, gnu11)
- License: GPLv3 since 4.2.2 (2007.10.7)
- Plugin support since 4.5 (2010.04.14), GPLv3 with runtime library exception
- GCC Resource Center at IITB (Indian Institute of Technology, Bombay)

- Compilation process is a pipeline, driven by the compiler driver
- C: preprocessor, compiler, assembler, linker
- Compiler: single process that
  - parses the source code into an Abstract Syntax Tree
  - verifies the AST
  - transforms the AST into an intermediate representation (IR)
  - optimizes the IR
  - transforms the IR into assembly

- GCC AST: language frontends produce GENERIC
  - Data structure: `tree`
  - Plugins can implement new attributes and pragmas, inspect structure declarations and variable definitions (gcc 4.6+)
- GCC IR #1: GIMPLE
  - Static Single Assignment (SSA) based representation
  - First set of optimization/transformation passes runs on GIMPLE (`-fdump-ipa-all`, `-fdump-tree-all`)
  - Data structures: `cgraph_node`, `function`, `basic_block`, `gimple`, `tree`
- GCC IR #2: RTL
  - GIMPLE is lowered to RTL (pre-SSA gcc had only this)
  - Second set of optimization passes runs on RTL (`-fdump-rtl-all`)
  - Data structures: `rtx`, `tree`

- machmode.def, tree.def, gimple.def, rtl.def
- machine modes: `VOIDmode`, `SImode`, `DImode`, `TImode`
- `tree` codes (~200 in 4.8): `ERROR_MARK`, `IDENTIFIER_NODE`, `INTEGER_TYPE`, `POINTER_TYPE`, `ARRAY_TYPE`, `RECORD_TYPE`, `VOID_TYPE`, `FUNCTION_TYPE`, `FUNCTION_DECL`, `FIELD_DECL`, `VAR_DECL`, `PARM_DECL`, `TYPE_DECL`, `COMPONENT_REF`, `ARRAY_REF`, `INDIRECT_REF`, `INTEGER_CST`, `STRING_CST`, etc
- gimple codes (~40 in 4.8): `GIMPLE_ASSIGN`, `GIMPLE_ASM`, `GIMPLE_CALL`, `GIMPLE_PHI`, `GIMPLE_NOP`, `GIMPLE_COND`, `GIMPLE_SWITCH`, `GIMPLE_RETURN`, etc
- rtl codes (~200 in 4.8): `MEM`, `REG`, `RETURN`, `CLOBBER`, `SET`, `BARRIER`, `INSN`, etc

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        return puts("hello world!\n");
}
```

- `gcc-4.8.1 -O2 -fdump-tree-all -fdump-ipa-all -fdump-rtl-all -fdump-passes`
- 97 SSA dumps
- 9 IPA dumps
- 57 RTL dumps

# -fdump-tree-ssa-raw

Listing 1: hello.c.016t.ssa

```
;; Function main (main, funcdef_no=24, decl_uid=2380, cgraph_uid=2

main (int argc, char * * argv)
{
  int _3;

  <bb 2>:
  gimple_call <puts, _3, "hello world!\n">
  gimple_return <_3>

}
```

# -fdump-tree-ssa

Listing 2: hello.c.016t.ssa

```
;; Function main (main, funcdef_no=24, decl_uid=2380, cgraph_uid=2

main (int argc, char * * argv)
{
  int _3;

  <bb 2>:
  _3 = puts ("hello world!\n");
  return _3;

}
```

- Loadable module system introduced in gcc 4.5
- Shared library loaded early right after command line parsing
- No well defined API, all public symbols available for plugin use
- Typical (intended :) use: new IPA/GIMPLE/RTL passes
  - Plugins can sign up for events, insert/remove/replace passes
  - No (easy) access to language frontends

# Comparison

- Related technologies: checkpatch.pl/coccinelle/sparse
- AST vs. GIMPLE/RTL
- Extra run vs. part of the regular compilation
- checkpatch.pl: no modification, source code analysis (pre-AST)
- sparse: no modification, only analysis
- coccinelle: modification by generating source patches $\rightarrow$ doesn't scale, harder to maintain

# Structure

- Some boilerplate code: `plugin_is_GPL_compatible`, `plugin_info`, `plugin_init`
- Pass registration: `register_callback`, `register_pass_info`, `simple_ipa_opt_pass`, `ipa_opt_pass_d`, `gimple_opt_pass`, `rtl_opt_pass`
- Callbacks: `PLUGIN_INFO`, `PLUGIN_START_UNIT`, `PLUGIN_PASS_MANAGER_SETUP`, `PLUGIN_ATTRIBUTES`, `PLUGIN_FINISH_TYPE`, `PLUGIN_FINISH_DECL`
- opt_pass: type, name, gate, execute, pass number, properties, todo flags

# Building

- C (4.5, 4.6, 4.7) vs. C++ (4.7, 4.8+)
  - Limited support for designated initializers in C++
- Cross-compilation: with the native compiler!
- `BUILDING_GCC_VERSION`, `GCCPLUGIN_VERSION` (since 4.7)
- No easy way to detect/depend on the target arch
  - __ARCH__ gives the wrong result for cross-compilation!
- Better plan: gcc-plugin-compat.h

# Overview

- Automatic constification of ops structures (200+ types in linux)
  - Structures with function pointer members only
  - Structures explicitly marked with a `do_const` attribute
- `no_const` attribute for special cases
  - Unfortunately many ops structures want to be written at runtime
- Local variables not allowed (compiler error generated)

# CONSTIFY

- PLUGIN_ATTRIBUTES callback: registers do_const and no_const attributes
  - Linux code patched by hand
  - Could be automated (static analysis, LTO)
- PLUGIN_FINISH_TYPE callback: sets TYPE_READONLY and C_TYPE_FIELDS_READONLY on eligible structure types
  - Only function pointer members, recursively
  - do_const is set, no_const is not set
- End result is that the frontend will do the dirty job of enforcing C variable constness
- GIMPLE pass: constified types cannot be used for local variables (stack is writable :)

## Structure Constification (CONSTIFY)

```
static bool constifiable(tree node) {
  tree field = TYPE_FIELDS(node);
  for (; field; field = TREE_CHAIN(field)) {
    fieldtype = TREE_TYPE(field);
    if (TREE_CODE(fieldtype) == POINTER_TYPE &&
        TREE_CODE(TREE_TYPE(fieldtype)) == FUNCTION_TYPE)
      continue;
    if (TREE_CODE(fieldtype) == RECORD_TYPE &&
        constifiable(fieldtype))
      continue;
    return false;
  }
  return true;
}
```

## Structure Constification (CONSTIFY)

```
static void constify_type(tree type)
{
  TYPE_READONLY(type) = 1;
  C_TYPE_FIELDS_READONLY(type) = 1;
  TYPE_CONSTIFY_VISITED(type) = 1;
}
```

## Structure Constification (CONSTIFY)

```
unsigned int i; tree var;
FOR_EACH_LOCAL_DECL(cfun, i, var) {
  tree type = TREE_TYPE(var);

  if (is_global_var(var))
    continue;

  if (TREE_CODE(type) != RECORD_TYPE)
    continue;

  if (!TYPE_READONLY(type) || !C_TYPE_FIELDS_READONLY(type))
    continue;

  if (!TYPE_CONSTIFY_VISITED(type))
    continue;

  error_at(DECL_SOURCE_LOCATION(var),
           "constified variable %qE cannot be local",
           var);
}
```

mm/shmem.c:1371:30: error: constified variable 'bad_file_operations' cannot be local

# Overview

- Goal: extract entropy from kernel state during boot
- Inspired by `https://factorable.net/`
- USENIX Security Symposium, August 2012
- Problem: much less entropy after boot than needed
- Result: vulnerable RSA and DSA keys used for SSH/TLS
- Some fixes in Linux but can we do better?

# LATENT_ENTROPY 2012

- Idea: compute a hash-like function embedded in the control flow graph of kernel boot code
- Similar and also simpler approach already in Phrack 66
- Insert a random combination of ADD/XOR/ROL insns into every basic block
- Mix end state into a global variable in the function epilogues
- Feed global variable (entropy) into the kernel entropy pools after each initcall
- Entropy is not actually accounted for until someone cryptanalyzes this whole thing :)
- More info on our mailing list

# LATENT_ENTROPY 2013

- Major change: keep gathering entropy after init
- During fork: fd table and vma list copying (variable loops)
- Module init
- All irq and softirq handlers (lots of loops)
  - Would be nice to use a percpu variable, but it's too arch dependent to be usable from a plugin
- Still no entropy accounting

# LATENT_ENTROPY

- `PLUGIN_START_UNIT` callback:
  - Fake the declaration of `extern volatile u64 latent_entropy`
  - Avoids patching in #include everywhere
- `PLUGIN_ATTRIBUTES` callback: registers `latent_entropy` attribute
  - Manually instrumented `__init` section definition, a few non-init functions
- `PLUGIN_PASS_MANAGER_SETUP`: registers core instrumentation logic
  - GIMPLE pass, invoked very late
  - Avoids interference with other optimizations, DCE in particular

## Latent Entropy Extraction (LATENT_ENTROPY)

```
// extern volatile u64 latent_entropy
gcc_assert(TYPE_PRECISION(long_long_unsigned_type_node) ==
    64);
latent_entropy_type = build_qualified_type(
    long_long_unsigned_type_node, TYPE_QUALS(
    long_long_unsigned_type_node) | TYPE_QUAL_VOLATILE);
latent_entropy_decl = build_decl(UNKNOWN_LOCATION, VAR_DECL,
     get_identifier("latent_entropy"), latent_entropy_type);

TREE_STATIC(latent_entropy_decl) = 1;
TREE_PUBLIC(latent_entropy_decl) = 1;
TREE_USED(latent_entropy_decl) = 1;
TREE_THIS_VOLATILE(latent_entropy_decl) = 1;
DECL_EXTERNAL(latent_entropy_decl) = 1;
DECL_ARTIFICIAL(latent_entropy_decl) = 1;
DECL_INITIAL(latent_entropy_decl) = build_int_cstu(
    long_long_unsigned_type_node, get_random_const());
lang_hooks.decls.pushdecl(latent_entropy_decl);
```

## Latent Entropy Extraction (LATENT_ENTROPY)

```
static unsigned int execute_latent_entropy(void)
{
  basic_block bb;
  tree local_entropy;

  bb = ENTRY_BLOCK_PTR->next_bb;

  // instrument each BB with an operation on the local entropy
  while (bb != EXIT_BLOCK_PTR) {
    perturb_local_entropy(bb, local_entropy);
    bb = bb->next_bb;
  };

  // mix local entropy into the global entropy variable
  perturb_latent_entropy(EXIT_BLOCK_PTR->prev_bb, local_entropy);
}
```

## Latent Entropy Extraction (LATENT_ENTROPY)

```
static void perturb_local_entropy(basic_block bb, tree
    local_entropy)
{
  gimple_stmt_iterator gsi;
  gimple assign;
  tree addxorrol, rhs;
  enum tree_code op;

  op = get_op(&rhs);
  addxorrol = fold_build2_loc(UNKNOWN_LOCATION, op,
      unsigned_intDI_type_node, local_entropy, rhs);
  assign = gimple_build_assign(local_entropy, addxorrol);
  gsi = gsi_after_labels(bb);
  gsi_insert_before(&gsi, assign, GSI_NEW_STMT);
  update_stmt(assign);
}
```

get_op: PLUS_EXPR, BIT_XOR_EXPR, LROTATE_EXPR

## Latent Entropy Extraction (LATENT_ENTROPY)

```
static int __init set_reset_devices(char *str)
{
  reset_devices = 1;
  return 1;
}
```

## Latent Entropy Extraction (LATENT_ENTROPY)

```
set_reset_devices (char * str)
{
  long unsigned int temp_latent_entropy.139;
  long unsigned int local_entropy.138;

  <bb 3>:
  local_entropy.138_5 = 1972019764950439624;

  <bb 2>:
  local_entropy.138_6 = local_entropy.138_5 ^
      9860098824752198l2;
  temp_latent_entropy.139_3 ={v} latent_entropy;
  temp_latent_entropy.139_4 = temp_latent_entropy.139_3 +
      local_entropy.138_6;
  latent_entropy ={v} temp_latent_entropy.139_4;
  reset_devices = 1;
  return 1;
}
```

## Latent Entropy Extraction (LATENT_ENTROPY)

```
0000000000000000 <set_reset_devices>:
   0:       mov      0x0(%rip),%rdx          # latent_entropy
   7:       push     %rbp
   8:       movabs   $0x16f10744be1e5dac,%rax
  12:       movl     $0x1,0x0(%rip)          # reset_devices
  1c:       mov      %rsp,%rbp
  1f:       pop      %rbp
  20:       add      %rdx,%rax
  23:       mov      %rax,0x0(%rip)          # latent_entropy
  2a:       mov      $0x1,%eax
  2f:       btsq     $0x3f,(%rsp)
  35:       retq
```

# Overview

- Goal: reduce lifetime of data on process kernel stacks by clearing the stack on kernel->user transitions
  - Per-arch hooks in the low-level kernel entry/exit code
  - Moved `thread_info` off the stack
- Initially blind memset on the entire kernel stack (8 kbytes)
  - Too slow (unused part of the stack is cache cold)
- Refinement: detect/clear only the used part of the stack
  - Looks for memset pattern from stack bottom to top
  - Optimization: check only a certain length (cache line)
- Needs to record stack depth in functions with a big stack frame
  - Manual inspection and patching
  - Instrumentation by a gcc plugin

# STACKLEAK

- Idea: insert function call to `pax_track_stack` if local frame size is over a specific limit
  - `pax_track_stack` records deepest used kernel stack pointer
- Problem: frame size info is available at the last RTL pass only, too late to insert complex code like a function call
- New strategy: instrument every function first and remove unneeded instrumentation later
  - Also finds all (potentially exploitable :) `alloca` calls

# STACKLEAK

- GIMPLE pass: inserts call to `pax_track_stack` into every function prologue
  - unless alloca is in the first basic block
  - alloca is bracketed with a call to `pax_check_alloca` and `pax_track_stack`
- RTL pass: removes unneeded `pax_track_stack` calls
  - if the local frame size is below the limit
  - if alloca is not used

# STACKLEAK

- Special paths for ptrace/auditing
  - Low-level kernel entry/exit paths can diverge for ptrace/auditing and leave interesting information on the stack for the actual syscall code
- Problems: still considerable overhead, races, leaks from a single syscall still possible
- Solution: dual process kernel stack, one used only for copying to/from userland
  - Needs static analysis to find all local variables whose address is sunk into `copy*user`
  - New gcc plugin, LTO

# STRUCTLEAK

- Goal: forcibly initialize local variables that can be copied to userland
- Prompted by CVE-2013-2141 (do_tkill kernel stack leak)
- Idea: if a local structure variable has `__user` annotated fields then forcibly initialize it if it isn't already
- `PLUGIN_FINISH_TYPE` callback: sets `TYPE_USERSPACE` on interesting structure types
- `PLUGIN_PASS_MANAGER_SETUP`: core instrumentation logic
  - GIMPLE pass, invoked early

```
// enumarate all local variables
unsigned int i; tree var;

FOR_EACH_LOCAL_DECL(cfun, i, var) {
  tree type = TREE_TYPE(var);

  if (!auto_var_in_fn_p(var, current_function_decl))
    continue;

  // only care about structure types
  if (TREE_CODE(type) != RECORD_TYPE)
    continue;

  // if the type is of interest, examine the variable
  if (TYPE_USERSPACE(type))
    initialize(var);
}
```

```
static void initialize(tree var)
{
  basic_block bb;
  gimple_stmt_iterator gsi;
  tree initializer;
  gimple init_stmt;

  // build the initializer expression
  initializer = build_constructor(TREE_TYPE(var), NULL);

  // build the initializer stmt
  init_stmt = gimple_build_assign(var, initializer);
  gsi = gsi_start_bb(ENTRY_BLOCK_PTR->next_bb);
  gsi_insert_before(&gsi, init_stmt, GSI_NEW_STMT);
  update_stmt(init_stmt);
}
```

# Overview

- Detects integer overflows in expressions used as a size parameter: `kmalloc(count * sizeof...)`
- Written by Emese Révfy, extends spender's old idea (preprocessor trick)
- Initial set of functions/parameters marked by the `size_overflow` function attribute
- Walks use-def chains and duplicates statements using a double-wide integer type
- SImode/DImode vs. DImode/TImode
- Special cases: asm(), function return values, constants (intentional overflows), memory references, type casts, etc
- More in our blog

# SIZE_OVERFLOW 2012

- `PLUGIN_ATTRIBUTES` callback: `size_overflow` attribute, takes arbitrary arguments (size parameter index)
  - Only a handful of functions are marked by hand
  - Hash table lookup for the rest (could be automated with LTO)
- GIMPLE pass: `handle_function` enumerates all function calls looking for the `size_overflow` attribute (or hash table)
- `handle_function_arg` starts the real work
  - Manually walks the use-def chain of the given function argument
  - Walk forks on binary/ternary operations and phi nodes
  - Walk stops at asm/call stmts, function parameters, globals, memory references, constants, etc

# SIZE_OVERFLOW 2012

- When a walk stops, stmt duplication begins
  - New variable is created with `signed_size_overflow_type`
  - DImode or TImode (signed)
- When stmt duplication reaches the original function call, the duplicated result is bounds checked
  - Against `TYPE_MAX_VALUE`/`TYPE_MIN_VALUE`
  - Optimization: check omitted if the walk did not find any stmt that could cause an overflow

# SIZE_OVERFLOW 2013

- Range checks on certain narrowing casts to catch integer truncation bugs
  - Caught various info leaks
  - CVE-2013-0914 (sa_restorer leak between userland processes)
  - CVE-2013-2141 (do_tkill kernel stack leak)
- New IPA pass to be able to walk across functions within a translation unit
- Spender's idea: combine with STACKLEAK (stack poisoning) and USERCOPY (check poison in data to be copied to userland) and trinity (fuzzing)
- Tons of info leak bugs triggered, not always trivial find the source of the leak (kernel stack → kernel heap → userland)

# Overview

- Goal: prevent executing userland code on amd64
- Idea: set most significant bit in all function pointers before dereference
  - Userland addresses become non-canonical ones, GPF on any dereference
- GIMPLE pass: handles C function pointers (`execute_kernexec_fptr`)
- RTL pass: handles function return values (`execute_kernexec_retaddr`)

# KERNEXEC/amd64 helper plugin

- Two methods: `bts` vs. `or` (reserves `%r10` for bitmask)
- Compatibility vs. performance
- Special cases: vsyscall, assembly source, asm()
  - `kernexec_cmodel_check` to exclude code in vsyscall sections
  - Manual verification/patching
  - GIMPLE pass to reload `r10` when clobbered by asm()

# LTO plans

- CONSTIFY: find all non-constifiable types/variables
- REFCOUNT: find all non-refcount `atomic_t/atomic64_t` uses
- SIZE_OVERFLOW: walk use-def chains across function calls, eliminate the hash table
- STACKLEAK: find all local variables whose address sinks into `copy*user`
- USERCOPY: find all `kmalloc-*` slab allocations that sink into `copy*user`

http://pax.grsecurity.net
http://grsecurity.net
irc.oftc.net #pax #grsecurity