Erasing secrets from RAM

Sponsored by

Laurent Simon
lmrs2@cam.ac.uk
https://www.cl.cam.ac.uk/~lmrs2/
Talk outline

• Threat Model
• Detection of Secret Data in Memory
• Compiler Plugin for Stack Erasure
• Conclusion
Threat Model

- Non malicious program $P$ handling secrets and then allegedly erasing them
- Ideal: Give attacker access to all of a system's volatile memory and CPU state
- Large code base: peripherals, OS kernel, OS userspace libraries, userspace program $P$, compiler, virtualisation primitives, SMP reordering, etc
- This talk: give attacker access to userspace memory and CPU state of program $P$
In Practice

- Main concern: compiler optimization

```c
void sensitive_function(...) {
   u8 sensitive_buffer[KEY_MAX] = \"\0\";
   ...
   zeromem(sensitive_buffer,KEY_MAX);
}
```

- Call to `zeromem()` REMOVED?
Talk outline

- Threat Model
- Detection of Secret Data in Memory
- Compiler Plugin for Stack Erasure
- Conclusion
Code Analysis

- **Static Code Analysis**: based on source code
  - Cannot account for compiler optimizations
  - Does not account memory accesses due to ABI, calling conventions, register spills, etc

- **Dynamic Code Analysis**: runs actual binary
  - Virtually no false positives
  - Only one instance at a time, so need comprehensive set of unit tests to avoid false negatives
Taint Tracking

- Taint = 0 (non-sensitive) or 1 (sensitive)

- Assign taint sources from which data becomes tainted -- eg reading a key from file or dev-selected variable
Taint Propagation/Removal

- Direct flows (assignment)
  
  \[ \text{new\_var} = \text{anyVar} \ BinOp \ \text{tainted\_var} \]
  
  \[ BinOp = | & ^ + - / * \]

- Pointer arithmetic, ie table lookups with tainted index (AES, format conversions)

- One-way functions remove taint, eg encryption functions
Resulting tool: secretgrind

- [https://github.com/lmrs2/secretgrind](https://github.com/lmrs2/secretgrind)

- Based on Valgrind, a runtime instrumentation framework

- Evaluation (work in progress) on gpg, openssl, mbedTLS
Preliminary Results

- Compiler optimizes out zeroing function: not encountered in our tests

- Developers' mistakes: eg forget to erase variables on both stack and heap
Preliminary Results (2)

- IO APIs' caching optimizations lead to (sublte) problems
- Example: read the first line of a PEM file to detect if it's a private or public key file
  - fread(size=LINE_SIZE,buf), OR buf=mmap(LINE_SIZE)
    
    \[ \ldots \]
    
    zeromem(buf,LINE_SIZE)

- Does not erase memory!
Preliminary Results (3)

- Formatting functions (*printf, *scanf, etc) prone to leaving copies of sensitive data on stack

- Recursive functions prone to leaving residual sensitive data on stack

- Calling conventions, ABI, register spills (GP or extended registers), leave residual data on stack
Talk outline

- Threat Model
- Detection of Secret Data in Memory
- Compiler Plugin for Stack Erasure
- Conclusion
Goal

- Automatically erase used stack and registers upon returning from functions annotated as "sensitive" by developer
- Integration with legacy code with minimal effort thru function annotation
- Low performance overhead
- Little or no support from kernel
- Clang/LLVM plugin for C, x86_64, single-threaded support for now
The OS and ABI

• Code provided by kernel for optimization (VDSO): typically time(), gettimeofday()

• Code provided by OS: libc and loader/linker

• Signal handlers: kernel saves CPU state on userspace stack before calling handler

• Use or not of frame pointer (%rbp) introduced in AMD64 ABI
The Compiler

- Compiler runtime: a library of optimized functions, eg `__udiv*` function for division
- Drop-in replacement functions, eg optimized `memset()`
- Linker: stubs added for function resolution
- Other optimizations: call tail optimization, shrink wrapping, etc
The Developer

- Variable-sized objects on stack, eg

  void foo(size_t len) {
      char buf[len];
      ...
  }

- `sigaltstack()` can change the stack for signal handler
Solution 1: Function Based

Erase used registers and stack on return of each function
Solution 1: Function Based

- Function-based, with signal: 239%
- Function-based, no signal: 86%
Solution 2: Stack Based

- Instrument every function to keep track of stack pointer (%rsp) in a global variable

- In annotated function only:
  - Erase used stack according to value in global variable
  - Erase all platform registers
Solution 3: Call Graph Based

- At compilation time, use the call graph for annotated functions
- Compute the maximum stack that can be used
- In annotated function only:
  - Erase (maximum) stack
  - Erase all used registers in call graph
Solution 3: Call Graph Based

- No performance hit besides actual zeroing
  But ...
- Breaks concept of shared library, better for embedded systems or statically linked programs
- No support for cycles, recursive functions, non-deterministic call graph (eg asynchronous events, eg posix AIO)
Talk outline

- Threat Model
- Detection of Secret Data in Memory
- Compiler Plugin for Stack Erasure
- Conclusion
Conclusion

- A new tool called secretgrind to test your software for erasure problems. Available as at https://github.com/lmrs2/secretgrind
- Plugin for Clang/LLVM for automated stack/register erasure. Code not released yet. But...
- Plugin is a hack and is fragile due to complexity of platforms and compilers, eg SafeStack
- What is the best way forward? Programming language, ABI/calling conventions, kernel, etc?
Thanks!

Questions?

Secretgrind beta available at:
https://github.com/lmrs2/secretgrind

Laurent Simon
lmrs2@cam.ac.uk
https://www.cl.cam.ac.uk/~lmrs2/