LINUX PLUMBERS CONFERENCE September 20-24, 2021

Security Improvements in GCC

Qing Zhao (qing.zhao@oracle.com)
Compilers and Toolchain Team
Oracle Linux Engineering and Virtualization

Agenda

- the security wishlist and the two new features to GCC.
- “call-used registers wiping on return” in GCC11.
- “stack variables auto-initialization” in GCC12.
- Future work.

The Security Wishlist for GCC

	gcc	clang
Speculative load hardening	no	yes
call-used registers wiping on return	yes (gcc11)	yes
stack variable auto-initialization	yes (gcc12)	yes
structure layout randomization	plugin in linux kernel	no
signed overflow protection	yes, usability issues	Yes, usability issues
unsigned overflow protection	no	Yes, usability issues
backward edge CFI	hardware only	hardware w/ arm64 soft
forward edge CFI	hardware only	Yes

ref to https://outflux.net/slides/2020/lpc/gcc-and-clang-security-feature-parity.pdf

Call-used Registers Wiping

outline

- Motivation
- Important questions and answers
- New feature added to GCC11
- Status of the patch
- A summary of the implementation
- Acknowledgment

Call-used Registers Wiping

Motivation

- Two major purposes:
  - Mitigating ROP (Return-oriented programming)
  - Preventing Information leakage through registers

Call-used Registers Wiping

Motivation (Mitigating ROP)

Features of ROP

- One of the ***most popular*** code reuse attack techniques.
- Execute ***gadget chains*** to perform malicious tasks.
- Each ***gadget*** typically ends in a ***return*** instruction,and located in a subroutine within the ***existing*** program.
- The destination of using ***gadget*** chains usually ***call system functions*** to perform malicious behavior.
- The ***registers*** usually are used to ***pass parameters*** for these system functions.

Therefore, cleaning the ***scratch registers*** that are used to ***pass parameters*** at ***return*** instructions should effectively mitigate ROP attack.

*(Ref to **Clean the Scratch Registers:A Way to Mitigate Return-Oriented Programming Attacks)***
2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors

Call-used Registers Wiping

Motivation (preventing information leakage)

- One of the well known security techniques is stack and register erasure. Ensuring that on return from a function, no data is left on the stack or in registers.
- There is a separate patch addressing the stack erasure problem
- “call-used registers wiping on return” can address the register erasure problem at the same time.

Call-used Registers Wiping

Important Questions

- Q1: ***Which registers*** **should be cleaned at the return of the function?**

Answer: **the caller-saved, i.e, call-used, or call-clobbered registers.**

For *ROP mitigation* purpose, only the call-used registers that *pass parameters* need to be cleaned.

For *register erasure* purpose, *all* the call-used registers need to be cleaned.

we should provide *multiple levels of control* for users for different purposes and control the runtime overhead as much as possible.

Call-used Registers Wiping

Important Questions

- Q2: **Why *zeroing* the registers other than randomizing them?**

Answer:

1) Zero _tends_ to be the ***safest*** choice as it's less "useful" to be used as a size, index, or pointer.
2) Generated code for zeroes is ***faster and smaller*** than that for other non-zero constants.

Call-used Registers Wiping

New Features added to GCC11

- Add a command-line ***option***:

-fzero-call-used-regs=[skip|used-gpr-arg|used-arg|all-gpr-arg |all-arg|used-gpr|all-gpr|used|all]

- Add a function ***attribute***:

zero_call_used_regs("skip|used-gpr-arg|used-arg|all-gpr-arg |all-arg|used-gpr|all-gpr|used|all")

skip: none
used: only used
all: all

gpr: general-purpose registers
arg: registers that can pass parameters

Multiple-level control

Call-used Registers Wiping

Status of the Patch

- ***Committed*** into ***GCC11*** on ***Oct. 2020.***
- ***X86*** and ***aarch64*** were ***fully*** supported, an ***optimized*** implementation for ***X86*** was provided.
- ***SPARC*** was supported ***later*** by defining the ***target hook*** TARGET_ZERO_CALL_USED_REGS on ***Dec. 2020.***
- The ***Linux kernel*** is being configured to use this feature to improve kernel's security on ***July 2021.***

Call-used Registers Wiping

Summary of the Patch

- ***Add a new pass*** in the beginning of "late_compilation", called "***pass_zero_call_used_regs***".
- In this new pass "pass_zero_call_used_regs":

  scan the exit block from backward to look for "***return***":

1) for each return, compute the "***need_zeroed_hardregs***" based on the user request, the data flow information, and function ABI information.
2) pass this "need_zeroed_hardregs" to the ***target hook*** "***zero_call_used_regs***" to generate the instruction sequence that zero the registers.

- The ***default implementation*** for the new ***target hook*** "***zero_call_used_regs***" will generate a sequence without any target-specific optimizations.
- ***X86*** backend has an ***optimized*** implementation.

Call-used Registers Wiping

Aknowlegement

- ***H.J.LU***: for the initial X86 implementation and bug fixes;
- ***Richard Sandiford***: lots of help during design phase and middle-end implementation;
- ***Segher Boessenkool***: provided many helpful insight during design phase;
- ***Uros Bizjak***: helped and reviewed on X86 implementation;
- ***Eric Botcazou***: made it work on SPARC;
- ***Kees Cook***: supported from Linux Kernel side;
- ***And all others*** who provided helpful insight during the discussion and code review….

Stack Variables Auto-initialization

outline

- Motivation
- New feature added to GCC12
- Status of the patch
- A summary of the implementation
- Acknowledgment

Stack Variables Auto-initialization

Motivation (1)

- Impact of uninitialized stack variables
  - Incorrect computations;
  - Change the program behavior unpredictably;
  - Enable malicious code execution;
  - Cause race condition if a lock variable check passes when it should not;
  - Etc...

Stack Variables Auto-initialization

Motivation (2)

- Tools that detect uninitialized stack variables:
  - ***Static*** tools
    - Both ***GCC and CLANG*** provides ***-Wuninitialized***
    - ***Visual studio*** provides an analysis plugin
  - ***Dynamic*** tools
    - Both ***GCC and CLANG*** provides ***-fsanitize=address/memory***
    - ***Valgrind***: https://valgrind.org

Stack Variables Auto-initialization

Motivation (3)

- Limitations of the tools:
- ***static*** tools:

1) Mostly base on ***Intra-procedural*** analysis, ***assume*** that the called function initializes every parameter;
2) Inside a function, has several limitations for uninitialized array elements, pointers, etc;
3) Potential infeasible paths due to conditional expressions that cannot be evaluated statically;

   static analysis tools have major issue: **false positives.**

- ***dynamic*** tools:

1) ***False negatives.*** along with occasional ***false positives.***

2) ***Runtime overhead***, ***size of logs*** are also big concerns.

Stack Variables Auto-initialization

Motivation (4)

- Both ***Microsoft*** compiler and CLANG (***APPLE*** and ***GOOGLE***) support pattern/zero init already;

http://lists.llvm.org/pipermail/cfe-dev/2020-April/065221.html

https://msrc-blog.microsoft.com/2020/05/13/solving-uninitialized-stack-memory-on-windows/

- ***pattern-***init is used in ***development*** build for ***debugging*** purpose;

  zero-init is used in ***production*** build for ***security*** purpose.

Stack Variables Auto-initialization

New Features added to GCC12

• ***Add*** a new GCC ***option***:
-ftrivial-auto-var-init=[uninitialized|pattern|zero], The ***default*** is '***uninitialized***'.
'***uninitialized***' doesn't initialize any automatic variables.
'***pattern***' Initialize with values which will likely transform logic bugs into crashes.
'***zero***' Initialize with zeroes.

• ***Add*** a new ***attribute*** for variable:
__attribute__*((uninitialized))*
the marked variable is uninitialized intentionally for performance purpose.

• ***Keep*** the current ***static warning on uninitialized*** variables untouched
in order to avoid "forking the language".

Stack Variables Auto-initialization

Status of the patch

- ***Committed*** to GCC12 on 9/9/2021.
- Some ***bugs filed and fixed*** after the above commit:
  - ICE with -ftrivial-auto-var-init=zero and zero size array (***PR102269***)
  - ICE in expand_DEFERRED_INIT, at internal-fn.c:3024 (***PR102273***)
  - ICE in can_native_interpret_type_p at gcc/fold-const.c:8800 (***PR102360***)

- Some ***bugs filed and not fixed*** yet:
  - New flag -ftrivial-auto-var-init=zero causes crash in pr82421.c (***PR102285***)
  - -ftrivial-auto-var-init=zero causes ice(***PR102281***)
  - -ftrivial-auto-var-init fails to initialize a variable, causes a spurious warning (***PR102276***)
  - ICE gimplification failed (***PR102359***)

Stack Variables Auto-initialization

A summary of the patch (1)

- Three Major Requirements:

1) ***all*** auto-variables that do not have an initializer should be initialized by this option, including the structure paddings.

2) ***keep*** the current static warnings on uninitialized variables untouched.

3) ***minimum*** run-time overhead.

Stack Variables Auto-initialization

A summary of the patch (2)

- The key of the design:

  Introduce a ***new internal function*** .DEFERRED_INIT to represent the initialization:

  - ***3 attributes:*** CONST, LEAF and NOTHROW;
  - ***3 parameters***: SIZE of the DECL, INIT_TYPE, IS_VLA

```
DEF_INTERNAL_FN (DEFERRED_INIT, ECF_CONST | ECF_LEAF | ECF_NOTHROW, NULL)

DECL = DEFERRED_INIT (SIZE of the DECL, INIT_TYPE, IS_VLA)
```

Stack Variables Auto-initialization

A summary of the patch (3)

- ***Add internal calls*** to .DEFFERED_INIT during ***gimplification***;
- ***Expand*** the .DEFFERED_INIT calls during ***expand phase*** to real initializations.
- ***Adjust uninitialized variable analysis*** with the new defs of .DEFFERED_INIT to maintain the uninitialized warnings.
- ***Adjust scalar replacement of aggregates*** with the new calls to .DEFFERED_INIT to minimize stack usage.

Stack Variables Auto-initialization

Acknowledgment

- ***Richard Biener***: lots of help during design and implementation;
- ***Richard Sandiford***: provided the initial idea of .DEFERRED_INIT and many help during the implementation;
- ***Kees cook***: supported from linux kernel side;
- ***Martin Jambor***: helped and reviewed on tree-sra.c;
- ***All others*** who helped during the long discussion and fixed the bugs;

Future Works
Call-used Register Wiping

- ***Two open bugs*** need to be fixed first

1) (X86) Adjust -fzero-call-used-regs to always use XOR (PR101891)

2) (arm) ICE: in df_exit_block_bitmap_verify, at df-scan.c:4164 with -mthumb -fzero-call-used-regs=used (PR100775)

- ***Some potential performance issues*** might need to be addressed when needed.

Future Works

Stack variable auto initialization

- One known issue need to be addressed first:

  Missing -Wuninitialized warning for address taken variables

- Bugs need to be fixed;
- Some more potential correctness and performance bugs might need to be addressed when needed.

Thank you!

Q & A

qing.zhao@oracle.com

Extra Slides

Stack Variables Auto-initialization

A summary of the patch (4)

Gimplification:

- For each auto-variable that ***does not have an explicit initializer,*** insert an initializer as:

X = DEFERRED_INIT (size of X, INIT_TYPE, IS_VLA)

if ***INIT_TYPE==PATTERN***, insert a call to ***__builtin_clear_padding (&X, 0)*** to initialize the ***paddings*** to zeros.

- For each auto-variable that ***has an explicit initializer***, insert a call to ***__builtin_clear_padding (&X, 0)*** to initialize the paddings.

Stack Variables Auto-initialization

A summary of the patch (5)

- ***Uninitialized variable analysis:***

Treat all defs with call to .DEFERRED_INIT as undefined,
to keep the uninitialized warnings.

- ***Scalar replacement of aggregates (SRA):***

Handle calls to .DEFERRED_INIT specifically as following:
tmp = .DEFERRED_INIT (size of tmp, INIT_TYPE, IS_VLA)
tmp is an aggregate, and this can be split-ted to the individual components as following:
tmp$0 = .DEFERRED_INIT (size of tmp$0, INIT_TYPE, IS_VLA);
tmp$1 = .DEFERRED_INIT (size of tmp$1, INIT_TYPE, IS_VLA);
tmp$2 = .DEFERRED_INIT (size of tmp$2, INIT_TYPE, IS_VLA);
to reduce the stack usages.

Stack Variables Auto-initialization

A summary of the patch (6)

- ***RTL expanding:***

X = DEFERRED_INIT (SIZE of X, INIT_TYPE, IS_VLA);

Block initialize X with zero/pattern according to its second argument ***INIT_TYPE***:

1) ***AUTO_INIT_ZERO***, use ***zeroes***;
2) ***AUTO_INIT_PATTERN***, use ***0xFE*** byte-repeatable pattern;

The variable X is initialized ***including paddings.***

WHY choose ***0xFE*** for ***pattern*** initialization:

- It is a non-canonical virtual address on x86_64, and at the high end of the i386 kernel address space.
- It is a very large float value (-1.694739530317379e+38).
- It is also an unusual number for integers.

Call-used Registers Wiping

The Complete Discussion History

https://gcc.gnu.org/pipermail/gcc-patches/2020-May/545075.html

https://gcc.gnu.org/pipermail/gcc-patches/2020-July/550018.html

https://gcc.gnu.org/pipermail/gcc-patches/2020-September/553212.html

https://gcc.gnu.org/pipermail/gcc-patches/2020-September/554771.html

https://gcc.gnu.org/pipermail/gcc-patches/2020-September/555119.html

https://gcc.gnu.org/pipermail/gcc-patches/2020-October/555619.html

https://gcc.gnu.org/pipermail/gcc-patches/2020-October/557568.html

Stack Variables Auto-initialization

Complete Discussion History

https://gcc.gnu.org/pipermail/gcc-patches/2021-February/565581.html

https://gcc.gnu.org/pipermail/gcc-patches/2021-March/567262.html

https://gcc.gnu.org/pipermail/gcc-patches/2021-May/570208.html

https://gcc.gnu.org/pipermail/gcc-patches/2021-July/574642.html

https://gcc.gnu.org/pipermail/gcc-patches/2021-July/575977.html

https://gcc.gnu.org/pipermail/gcc-patches/2021-July/576072.html

https://gcc.gnu.org/pipermail/gcc-patches/2021-July/576341.html

https://gcc.gnu.org/pipermail/gcc-patches/2021-August/576994.html