УЯЗВИМОСТИ GCC И LLVM К АТАКАМ НА КОНВЕЙЕР ОПТИМИЗАЦИИ

Муравьёв С. К.[1]

DOI: 10.21681/2311-3456-2025-4-9-16

Цель исследования: *выработка рекомендаций по реализации средств разработки безопасного программного обеспечения (РБПО) и внедрению процессов РБПО на основе анализа угроз безопасности информации при разработке программного обеспечения, связанных с возможностью создания и использования злоумышленниками модулей расширения для оптимизирующих компиляторов.*

Метод(ы) исследования: *к основным методам исследования относятся анализ и синтез, моделирование и эксперимент.*

Результат(ы) исследования: *на основании анализа действующих нормативных требований к разработчикам безопасного программного обеспечения в части оценки угроз безопасности информации со стороны средств разработки программного обеспечения определены актуальность, цель и предмет исследования.*

Рассмотрены особенности анализа и трансформации исходного кода оптимизирующими компиляторами GCC и LLVM в процессе оптимизации исходного кода. Показана уязвимость таких компиляторов к вредоносным вмешательствам в конвейер оптимизации, которые могут быть осуществлены злоумышленниками через штатные программные интерфейсы, предусмотренные для повышения функциональности таких компиляторов и эффективности разрабатываемого с их помощью программного обеспечения.

Продемонстрирована возможность практической реализации атак, которые меняют конвейер оптимизации таким образом, что алгоритм функционирования целевого приложения принципиально меняется требуемым злоумышленнику образом. При этом такие атаки не связаны с нарушением целостности и конфиденциальности исходного кода целевого приложения и исполняемых файлов средств разработки программного обеспечения.

Проанализированы сложности обнаружения подобных угроз и определены способы их нейтрализации. В итоге выработаны рекомендации, которые могут быть использованы при проектировании и реализации безопасных компиляторов и безопасного программного обеспечения, а также при внедрении соответствующих процессов РБПО.

Научная новизна: *показана необходимость расширения нормативных требований к безопасным компиляторам языков С/С++ и мер по разработке безопасного программного обеспечения, в части необходимости контроля использования модулей расширения для применяемых инструментальных средств.*

Ключевые слова: *средства разработки, программное обеспечение, оптимизирующий компилятор, угроза безопасности информации, РБПО.*

Введение

Разработке безопасного программного обеспечения (РБПО) в настоящее время уделяется значительное внимание [1], что находит своё отражение в активном развитии соответствующей нормативно-правовой[2] и нормативно-технической документации, определяющей общие требования к содержанию и порядку выполнения работ, связанных с РБПО[3], угрозы безопасности информации при разработке программного обеспечения (ПО)[4], а также общие требования к безопасным компиляторам языков С/С++[5]. При этом, среди прочего, данные стандарты определяют актуальность исследования угроз безопасности информации, возникающих в ходе разработки ПО со стороны средств разработки.

В России в качестве основы для реализации инструментальных средств разработки на языках программирования С и С++, реально можно рассматривать лишь экосистемы GCC и LLVM, включающие в свой состав современные функциональные оптимизирующие компиляторы. Алгоритм работы таких компиляторов может стать причиной внедрения в разрабатываемое ПО уязвимостей, которые могут создавать серьёзные угрозы безопасности для созданного с их помощью программного обеспечения. Одна из особенностей алгоритмов их работы заключается в возможности произвольного изменения процессов оптимизации исходного кода с помощью внешних модулей, подключаемых через штатные

1 Муравьёв Сергей Константинович, кандидат технических наук, начальник отдела ООО НТП «Криптософт». Пенза, Россия. E-mail: smurav@mail.ru

2 Приказ ФСТЭК России от 01 декабря 2023 г. № 240 «Об утверждении Порядка проведения сертификации процессов безопасной разработки программного обеспечения средств защиты информации».

3 ГОСТ Р 56939 – 2024. Защита информации. Разработка безопасного программного обеспечения. Общие требования.

4 ГОСТ Р 58412 – 2019. Защита информации. Разработка безопасного программного обеспечения. Угрозы безопасности информации при разработке программного обеспечения.

5 ГОСТ Р 71206 – 2024. Защита информации. Разработка безопасного программного обеспечения. Безопасный компилятор языков С/С++. Общие требования.

программные интерфейсы [2], доступ к которым должен быть ограничен для нарушителя [3,4]. Таким образом, исследования уязвимостей GCC и LLVM к атакам на конвейер оптимизации, которым и посвящено данное исследование, обладают несомненной актуальностью.

Особенности работы компиляторов GCC и LLVM

Оптимизирующие компиляторы, входящие в состав GCC и LLVM, применяют к исходному коду множество различных операций для анализа и трансформации, называемые проходами. Компиляторы предоставляют сотни готовых проходов для анализа и трансформации исходного кода, которые в каждом из компиляторов называются по-своему и имеют различное назначение. При этом в зависимости от типа и версии компилятора эти проходы объединяются в несколько типовых конвейеров оптимизации, которые также могут содержать сотни различных проходов, как показано на (рис. 1) и (рис. 2), которые дополнительно можно изменить или расширить с помощью модулей расширения, подключаемых через штатные программные интерфейсы.

Большинство высококвалифицированных разработчиков, повседневно применяющих данные конвейеры, даже не представляют назначение отдельных операций и не контролируют фактический состав конвейера оптимизации при каждом запуске процесса компиляции. При наличии небезопасных программных интерфейсов для изменения конвейеров оптимизации и отсутствии эффективных средств контроля их безопасности создаются условия для проведения вредоносных вмешательств в процесс компиляции исходного кода.

```
g++-14 -O2 rnd.cpp -o rnd -fdump-passes
```

```
*warn_unused_result                    : ON
*diagnose_omp_blocks                   : OFF
*diagnose_tm_blocks                    : OFF
tree-omp_oacc_kernels_decompose        : OFF
tree-omplower                          : ON
tree-lower                             : ON
tree-tmlower                           : OFF
tree-ehopt                             : ON
tree-eh                                : ON
tree-coro-lower-builtins               : OFF
tree-cfg                               : ON
*warn_function_return                  : ON
tree-coro-early-expand-ifns            : OFF
tree-ompexp                            : ON
*build_cgraph_edges                    : ON
*free_lang_data                        : ON
ipa-visibility                         : ON
скрыто 360 строк!
```

Рис. 1. Пример типового конвейера оптимизации GCC

На примере простейшего тестового приложения, написанного с использованием языка программирования C++, которое формирует несколько числовых последовательностей с помощью различных стандартных генераторов случайных чисел [5], можно продемонстрировать возможность создания вредоносных подключаемых модулей для компиляторов из состава GCC и LLVM, которые способны изменить процессы инициализации генераторов случайных чисел таким образом, что злоумышленник сможет

```
opt rnd.ll -passes="default<O2>" -o /dev/null -print-pipeline-passes
```

```
annotation2metadata,forceattrs,inferattrs,coro-early,function<eager-inv>(ee-instrument<>,lower-expect,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;no-switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch;no-speculate-unpredictables>,sroa<modify-cfg>,early-cse<>),openmp-opt,ipsccp,called-value-propagation,globalopt,function<eager-inv>(mem2reg,instcombine<max-iterations=1;no-use-loop-info;no-verify-fixpoint>,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch;no-speculate-unpredictables>),always-inline,require<globals-aa>,function(invalidate<aa>),require<profile-summary>,cgscc(devirt<4>(inline,function-attrs<skip-non-recursive-function-attrs>,openmp-opt-cgscc,function<eager-inv;no-rerun>(sroa<modify-cfg>,early-cse<memssa>,speculative-execution<only-if-divergent-target>,jump-threading,correlated-propagation,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch;no-speculate-unpredictables>,instcombine<max-iterations=1;no-use-loop-info;no-verify-fixpoint>,aggressive-instcombine,libcalls-shrinkwrap,tailcallelim,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch;no-speculate-unpredictables>,reassociate,constraint-elimination,loop-mssa(loop-instsimplify,loop-simplifycfg,licm<no-allowspeculation>,loop-rotate<header-duplication;no-prepare-for-lto>,licm<allowspeculation>,simple-loop-unswitch<no-nontrivial;trivial>),simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch;no-speculate-unpredictables>,instcombine<max-iterations=1;no-use-loop-info;no-verify-fixpoint>,loop(loop-idiom,indvars,simple-loop-unswitch<no-nontrivial;trivial>,loop-deletion, ....
скрыто 100 элементов!
```

Рис. 2. Пример типового конвейера оптимизации LLVM

полностью предсказать результаты работы такого приложения. При этом такая атака не связана с эксплуатацией ненадёжности рассматриваемых генераторов псевдослучайных чисел [6], нарушением конфиденциальности и целостности исходного кода, а также целостности исполняемых файлов инструментальных средств разработки.

Ключевой участок исходного кода тестового приложения, который отвечает за получение набора случайных чисел с равномерным распределением, приведен на (рис. 3). Генератор случайных чисел инициализируется случайным значением перед получением каждого следующего числа. Приложение исполняет ключевой участок кода для десяти различных генераторов случайных чисел, входящих в состав стандартной библиотеки языка C++. Результат исполнения тестового приложения приведён на (рис. 4).

rnd.cpp

```
16  EngineType engine{};
17  static random_device rd{};
18  static uniform_int_distribution val{1, 9};
19
20  for(int j{0}; j < 10; ++j) {
21      engine.seed(rd());
22      cout << val(engine) << " ";
23  }
24  cout << endl;
```

Рис. 3. Ключевой участок исходного кода тестового приложения

./rnd

```
8878213884
5251249699
9589477566
3594632529
1796481463
5872868523
6937872589
6786366816
8221184864
9324776171
```

Рис. 4. Результат исполнения тестового приложения

При наличии доступа к исходному коду злоумышленник может попытаться исследовать алгоритм работы приложения. Но реализация данной угрозы не позволит раскрыть реальные числовые последовательности из-за применения случайных чисел для инициализации генераторов.

Злоумышленник также может попытаться нарушить целостность исходного кода приложения и внедрить в него вредоносные правки или добиться нужного поведения приложения за счёт внедрения изменений непосредственно в компилятор или другие инструментальные средства, используемые при разработке. К счастью, в настоящее время существует довольно много различных средств защиты, которые позволяют оперативно обнаруживать факты нарушения целостности как исходного кода, так и исполняемых файлов, а также эффективно противодействовать подобным угрозам.

Однако, существует возможность проведения атаки, которая не связана с нарушением конфиденциальности и целостности исходного кода, а также целостности исполняемых файлов компилятора и других инструментальных средств разработки [7]. Подобная атака может быть реализована за счёт использования штатных интерфейсов компилятора для встраивания проходов оптимизации, которые способны изменить исходный алгоритм программы.

Перед применением конвейера оптимизации современные компиляторы выполняют преобразование исходного кода в промежуточное представление, в котором исходный код разбивается на базовые блоки, содержащие простые последовательности инструкций. При этом все переменные, в том числе и виртуальные, представляются в форме с единственным статическим присваиванием (SSA) [8].

Подключаемые расширения для компилятора имеют возможность целенаправленного поиска инструкций в промежуточном представлении, их удаления, изменения параметров или даже их замены на альтернативные инструкции, что может принципиально изменить исходный алгоритм работы компилируемого приложения.

Промежуточное представление GCC [9] напоминает упрощённый язык C, в котором раскрыты все высокоуровневые структуры данных и комплексные инструкции для управления потоком исполнения. На (рис. 5) представлен ключевой участок промежуточного представления исходного кода тестового приложения, сформированного GCC.

Промежуточное представление LLVM [10] больше напоминает язык Ассемблер и его значительно сложнее читать, т.к. вместо человекочитаемых развёрнутых сигнатур, используются декорированные (англ. mangled) имена. В данном представлении имена функций начинаются с символа @, а имена переменных заменяются на числа, перед которыми ставится символ %.

На (рис. 6) представлен ключевой участок промежуточного представления исходного кода тестового приложения, сформированного LLVM. Полный вариант промежуточного представления не приводится из-за его существенного объёма.

DOI: 10.21681/2311-3456-2025-4-9-16

```
g++-14 -O1 -fdump-tree-gimple rnd.cpp -o rnd...
cat ./rnd.cpp.005t.original
```

```
1112    {
1113    int j;
1115    j = 0;
1116    goto <D.89377>;
1117    <D.89376>:
1118    _5 = std::random_device::operator() (&rd);
1119    _6 = (long unsigned int) _5;
1120    std::linear_congruential_engine<long unsigned int, 16807, 0, 2147483647>::seed (&engine, _6);
1121    _7 = std::uniform_int_distribution<int>::operator()<std::linear_congruential_engine<...> > (&val, &engine);
1122    _8 = std::basic_ostream<char>::operator<< (&cout, _7);
1123    std::operator<< <std::char_traits<char> > (_8, " ");
1124    j = j + 1;
1125    <D.89377>:
1126    if (j <= 9) goto <D.89376>; else goto <D.89374>;
1127    <D.89374>:
1128  }
```

Рис. 5. Промежуточное представление GCC ключевого участка исходного кода

```
clang -S -emit-llvm -O1 rnd.cpp -o rnd.ll ...
cat ./rnd.ll
```

```
1085  define linkonce_odr dso_local void @_ZNSt26linear_congruential_... {
1086    %3 = alloca ptr, align 8
1087    %4 = alloca i32, align 4
1088    store ptr %0, ptr %3, align 8
1089    store i32 %1, ptr %4, align 4
1090    %5 = load ptr, ptr %3, align 8
1091    %6 = load i32, ptr %4, align 4
1092    call void @_ZNSt26...(ptr noundef nonnull align 4 1086 dereferenceable(4) %5, i32 noundef %6)
1093    ret void
1094  }
```

Рис. 6. Промежуточное представление LLVM части ключевого участка исходного кода

```
gcc/seedpass.cpp
```

```
45    FOR_ALL_BB_FN(bb, fun)
46    {
47            gimple_bb_info *bb_info = &bb->il.gimple;
50            for (gimple_stmt_iterator gsi = gsi_start(bb_info->seq); !gsi_end_p(gsi); gsi_next(&gsi)) {
52                    gimple *g = gsi_stmt(gsi);
53                    if (GIMPLE_CALL == g->code) {
55                            tree fn_call = gimple_call_fn(g);
56                            tree fn_decl = TREE_OPERAND(fn_call, 0);
57                            tree fn_id = DECL_NAME(fn_decl);
58                            const char *fn_name = IDENTIFIER_POINTER(fn_id);
59                            if (0 == strcmp("seed", fn_name)) {
62                                    tree mal_val = build_int_cst_type(integer_type_node, 1234567);
63                                    gimple_call_set_arg(g, 1, mal_val);
64                            }
65                    }
66            }
67    }
```

Рис. 7. Ключевой участок исходного кода расширения GCC

llvm/seedpass.cpp

```
71    for (auto &BB : F) {
72        for (auto &I : BB) {
73            CallInst *CI = dyn_cast<CallInst>(&I);
74            if (nullptr == CI)
75                continue;
77            Function *CF = CI->getCalledFunction();
78            if (nullptr == CF)
79                continue;
81            std::string FN = demangle(CF->getName());
82            if (FN.ends_with("::seed(unsigned long)")) {
84                Value *V = CI->getOperand(1);
85                Value *NV =  ConstantInt::get(V->getType(), 1234567, false);
86                CI->setArgOperand(1, NV);
87            }
88        }
89    }
```

Рис. 8. Ключевой участок исходного кода расширения LLVM

```
clang -O1 -fpass-plugin=./plugins/llvm_seedpass.so rnd.cpp ...
```

```
...
_ZNSt26linear_congruential_engineImLm48271ELm0ELm2147483647EE4seedEm => std::linear_congruential_engine<unsigned
long, 48271ul, 0ul, 2147483647ul>::seed(unsigned long)

_ZNSt23mersenne_twister_engineImLm32ELm624ELm397ELm31ELm2567483615ELm11ELm4294967295ELm7ELm2636928640EL
m15ELm4022730752ELm18ELm1812433253EE4seedEm => std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul,
2567483615ul, 11ul, 4294967295ul, 7ul, 2636928640ul, 15ul, 4022730752ul, 18ul, 1812433253ul>::seed(unsigned long)

_ZNSt23mersenne_twister_engineImLm64ELm312ELm156ELm31ELm13043109905998158313ELm29ELm61489146912365172O5EL
m17ELm8202884508482404352ELm37ELm18444473444759240704ELm43ELm6364136223846793005EE4seedEm =>
std::mersenne_twister_engine<unsigned long, 64ul, 312ul, 156ul, 31ul, 13043109905998158313ul, 29ul, 6148914691236517205ul,
17ul, 8202884508482404352ul, 37ul, 18444473444759240704ul, 43ul, 6364136223846793005ul>::seed(unsigned long)

_ZNSt26subtract_with_carry_engineImLm24ELm10ELm24EE4seedEm => std::subtract_with_carry_engine<unsigned long, 24ul,
10ul, 24ul>::seed(unsigned long)
_ZNSt26subtract_with_carry_engineImLm48ELm5ELm12EE4seedEm => std::subtract_with_carry_engine<unsigned long, 48ul, 5ul,
12ul>::seed(unsigned long)
...
```

Рис. 9. Примеры преобразования декорированных имён в расширении LLVM

Обратите внимание, что в строке 1120 промежуточного представления GCC и строке 1092 представления LLVM видно расширение сигнатуры метода seed[6]. У метода появился дополнительный параметр, который отсутствует в оригинальной сигнатуре, но необходим для передачи указателя this в рамках используемого соглашения о вызовах thiscall, применяемого при вызове методов классов для языка C++.

Атаки на конвейер оптимизации

Инфраструктуры GCC и LLVM предоставляют разработчикам программные интерфейсы для подключения модулей расширения, предназначенных, помимо прочего, для встраивания новых проходов оптимизации. Перед разработчиком такого расширения открываются широкие возможности по анализу и модификации промежуточного представления исходного кода.

На рис. 7 и рис. 8 показаны примеры ключевых участков исходного кода подключаемых расширений компиляторов из состава GCC и LLVM, соответственно, осуществляющих перебор базовых блоков компилируемого приложения с целью обнаружения вызовов функции инициализации генераторов случайных чисел. В случае обнаружения подобных вызовов они осуществляют замену любого входного значения, предусмотренного алгоритмом работы программы, на целочисленную константу, известную злоумышленнику. При этом для такой замены не требуется доступ к исходному коду, так как вредоносный проход может проверить и изменить вызовы различных

6 Std::linear_congruential_engine::seed. URL: https://cplusplus.com/reference/random/linear_congruential_engine/seed, (дата обращения: 03.10.2024).

```
g++-14 -fdump-tree-optimized -fplugin=./plugins/gcc_seedpass.so rnd.cpp -o rnd_gcc_instr
cat ./rnd_gcc_instr-rnd.cpp.265t.optimized
```

```
739     <bb 11> :
740     _43 = std::random_device::operator() (&rd);
741     _5 = _43;
742     _6 = (long unsigned int) _5;
743     std::linear_congruential_engine<long unsigned int, 16807, 0, 2147483647>::seed (&engine, 1234567);
744     _46 = std::uniform_int_distribution<int>::operator()<std::linear_congruential_engine<...> (&val, &engine);
745     _7 = _46;
746     _48 = std::basic_ostream<char>::operator<< (&cout, _7);
747     _8 = _48;
748     std::operator<< <std::char_traits<char> > (_8, " ");
749     j_50 = j_9 + 1;
```

Рис. 10. Промежуточное представление GCC после вредоносной оптимизации

```
opt --load-pass-plugin=./plugins/llvm_seedpass.so -passes="seedpass,default<O2>" rnd.ll -S -o rnd_opt.ll
cat ./rnd_opt.ll
```

```
522     27:                                       ; preds = %24
523       %28 = call noundef i32 @_ZNSt13random_deviceclEv(...)
524       %29 = zext i32 %28 to i64
525       call void @_ZNSt26linear_...seedEm(ptr noundef nonnull align 8 dereferenceable(8) %3, i64 noundef 1234567)
526       %30 = call noundef i32 @_ZNSt24uniform_...(...)
527       %31 = call noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(...)
528       %32 = call noundef nonnull align 8 dereferenceable(8) ptr _ZStlsISt11char_traits...(...)
529       br label %33
```

Рис. 11. Промежуточное представление LLVM после вредоносной оптимизации

функций. Таким образом, данный метод атаки может быть с успехом применён не только к генераторам случайных чисел, но и к совершенно другим задачам.

Необходимо отметить тот факт, что из-за особенностей применяемого соглашения о вызовах оба расширения заменяют второй аргумент метода, отвечающего за инициализацию генератора, несмотря на наличие лишь одного параметра в исходном определении метода.

Также стоит обратить внимание на то, что в алгоритме работы расширения LLVM выполняются дополнительные преобразования декорированных имён вызываемых функций, как показано на (рис. 9).

На (рис. 10) показано промежуточное представление исходного кода тестового приложения, полученное после выполнения вредоносного прохода оптимизации для GCC, а на (рис. 11) – для LLVM.

После такой вредоносной оптимизации каждый элемент числовой последовательности, формируемый тестовым приложением, будет известен злоумышленнику. При этом для стороннего наблюдателя такая числовая последовательность никак не будет отличаться от любой другой псевдослучайной последовательности, формируемой соответствующим генератором. Также следует отметить, что исполняемый файл тестового приложения с вредоносной оптимизацией, скомпилированный с помощью GCC, и вариант, скомпилированный с помощью LLVM, выдают одинаковые числовые последовательности, как показано на (рис. 12).

```
./rnd_instr

6666666666
7777777777
3333333333
9999999999
5555555555
6666666666
5555555555
6666666666
8888888888
6666666666
```

Рис. 12. Результат исполнения приложения с вредоносной оптимизацией

Защита от атак на конвейер оптимизации

Исключение возможности проведения описанных атак на конвейер оптимизации может быть достигнуто за счёт полного запрета внешних модулей расширения для безопасного компилятора или путём реализации дополнительного механизма контроля аутентичности подключаемых модулей. Также можно порекомендовать использовать в качестве платформы для безопасной разработки программного обеспечения такие защищённые операционные системы, как QP ОС [11], которые обеспечивают замкнутость

программной среды исполнения соответствующих инструментальных средств, что исключает возможность несанкционированного внедрения в процессы компиляции.

Если полный запрет модулей расширения или контроль их аутентичности невозможны, то необходимо уделить повышенное внимание вопросам журналирования и контроля процессов компиляции, что позволит выявить несанкционированные вмешательства в конвейер оптимизации. В общих требованиях к безопасному компилятору языков C/C++ присутствует требование по ведению базы данных (БД) компиляции, где должны фиксироваться информация, позволяющая идентифицировать задействованные программы, а также настройки и конфигурационные файлы этих программ.

К сожалению, общие требования не содержат указаний по поводу фиксации информации о внешних модулях, которые могут динамически подключаться к программам, задействованным в процессе трансляции. Поэтому необходимо расширить общие требования к безопасному компилятору языков C/C++ в части перечня сведений, подлежащих сохранению в БД компиляции. В такой БД дополнительно должен фиксироваться фактический состав применяемого конвейера оптимизации и контрольные значения всех подключаемых расширений компилятора. При этом должен быть определён белый список безопасных проходов оптимизации, разрешенных к применению.

Кроме того, в БД компиляции необходимо фиксировать информацию о системном окружении операционной системы, так как оно может оказывать неявное влияние на функционирование инструментальных средств разработки и обеспечивать возможность подмены существующих или подключения новых модулей расширения.

Выводы

В статье рассмотрены особенности анализа и трансформации исходного кода современными компиляторами из состава GCC и LLVM в процессе оптимизации исходного кода. Показана возможность практической реализации атак на конвейер оптимизации, способных принципиально изменить алгоритм функционирования компилируемого приложения, а также даны рекомендации по нейтрализации подобных угроз.

Разработчики и производители ПО могут использовать сведения, приведённые в статье, при проектировании и внедрении процессов РБПО, а также при разработке безопасного программного обеспечения.

Практическое подтверждение предлагаемых научных решений может быть выполнено путём исследования исходных кодов тестового приложения и всех рассмотренных расширений, а также инструкций по их сборке и применению, открытых и доступных для свободного использования[7].

7 Github repository. URL: https://github.com/smurav/passes, (дата обращения: 03.10.2024).

Литература

1. Арустамян С. С., Вареница В. В., Марков А. С. Методические и реализационные аспекты внедрения процессов разработки безопасного программного обеспечения // Безопасность информационных технологий. 2023. Т. 30. № 2. С. 23–37.
2. Наке К., Кван Э. LLVM 17: Инфраструктура для разработки компиляторов / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2024. – 370 с. – ISBN 978-5-93700-303-4.
3. Леонов Н. В. Противодействие уязвимостям программного обеспечения. Часть 1. Онтологическая модель // Вопросы кибербезопасности. № 2(60). 2024. DOI: 10.21681/2311-3456-2024-2-87-92.
4. Леонов Н. В. Противодействие уязвимостям программного обеспечения. Часть 2. Аналитическая модель и концептуальные решения // Вопросы кибербезопасности. 2024, № 3(61). С. 90–95. DOI: 10.21681/2311-3456-2024-3-90-95.
5. Дейтел П., Дейтел Х. C++20 для программистов. СПб.: Питер, 2024. – 1056 с. – ISBN 978-5-4461-2359-9.
6. Белов, А. А. Ненадежность известных генераторов псевдослучайных чисел / А. А. Белов, Н. Н. Калиткин, М. А. Тинтул // Журнал вычислительной математики и математической физики. – 2020. – Т. 60, № 11. – С. 1807–1814. – DOI 10.31857/S0044466920110046. – EDN CTJCWS.
7. Муравьёв, С. К. Угрозы безопасности информации со стороны модулей расширения для оптимизирующих компиляторов // Безопасность информационных технологий. – 2024. – Т. 31, № 4. – С. 44–55. – DOI 10.26583/bit.2024.4.02. – EDN LUNGGO.
8. Rastello F., Tichadou F. B. SSA-based Compiler Design. – Springer, 2022. – ISBN 978-3-030-80514-2. DOI: https://doi.org/10.1007/978-3-030-80515-9.
9. Khedler U. GCC Translation Sequence and Gimple IR. URL: https://reup.dmcs.pl/wiki/images/d/da/Gcc-gimple.pdf, (дата обращения: 02.10.2024).
10. Min-Yih H. LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries. – Packt Publishing, 2021. – ISBN 978-1-83882-495-2.
11. Егоров, В. Ю. Развитие операционной системы QP ОС // Новые информационные технологии и системы: Сборник научных статей по материалам XVII Международной научно-технической конференции, 18–19 ноября 2020 года. – Пенза: ПГУ, 2020. – С. 45–47. – EDN EJXOIX.

THE A VULNERABILITIES OF GCC AND LLVM TO OPTIMIZATION PIPELINE ATTACKS

Muravyev S. K.[8]

Keywords: *development tools, software, compiler, information security threat, GCC, LLVM.*

Purpose of the study: *the purpose of the work is to develop recommendations for the implementation of secure software development tools and the implementation development processes based on the analysis of information security threats in software development related to the possibility of creating and using extension modules for optimizing compilers by attackers.*

Methods of research: *the main research methods include analysis and synthesis, modeling and experiment.*

Result(s): *the article examines the vulnerabilities of optimizing compilers of GCC and LLVM to malicious interference in the optimization pipeline, which can be carried out by hackers through standard software interfaces provided to enhance the functionality of such compilers and the effectiveness of software developed with their help.*

The relevance of the work *is determined by the current regulatory and technical requirements for developers of secure software to analyze information security threats from software development tools, one of the key elements of which are optimizing compilers. The article discusses the features of the analysis and transformation of the source code by optimizing compilers of GCC and LLVM in the process of optimizing the source code. The possibility of practical implementation of attacks that change the optimization pipeline in such a way that the algorithm of functioning of the target application fundamentally changes in the way required by the attacker is shown. As a result, recommendations are given on how to neutralize such threats.*

Scientific novelty: *the paper shows the need to expand the regulatory requirements for secure C/C++ compilers and measures to develop secure software, in terms of the need to control the use of extension modules for the tools used.*

References

1. Arustamjan S. S., Varenica V. V., Markov A. S. Metodicheskie i realizacionnye aspekty vnedrenija processov razrabotki bezopasnogo programmnogo obespechenija // Bezopasnost' informacionnyh tehnologij. 2023. T. 30. № 2. S. 23–37.
2. Nacke K., Kwan A. Learn LLVM 17. A beginner's guide to learning LLVM compiler tools and core libraries with C++. Packt Publishing, 2024. – ISBN 978-1-83763-134-6.
3. Leonov N. V. COUNTERING SOFTWARE VULNERABILITIES. Part 1. ONTOLOGICAL MODEL // Voprosy kiberbezopasnosti. 2024, № 2(60). S. 87–92. DOI: 10.21681/2311-3456-2024-2-87-92.
4. Leonov N. V. COUNTERING SOFTWARE VULNERABILITIES. Part 2. ANALYTICAL MODEL AND CONCEPTUAL SOLUTIONS // Voprosy kiberbezopasnosti. 2024, № 3 (61). S. 90–95. DOI: 10.21681/2311-3456-2024-3-90-95.
5. Deitel P., Deitel H. C++20 for Programmers. Pearson, 2022. ISBN 978-0136905691.
6. Belov, A. A. Unreliability of Available Pseudorandom Number Generators / A. A. Belov, M. A. Tintul, N. N. Kalitkin // Computational Mathematics and Mathematical Physics. – 2020. – Vol. 60, No. 11. – P. 1747-1753. – DOI 10.1134/S0965542520110044.
7. Muravyev S. K. Information security threats of optimizing compilers' plugins. IT Security (Russia), [S.l.], v. 31, no. 4, p. 44–55, 2024. ISSN 2074-7136. ISSN 2074-7136. DOI: http://dx.doi.org/10.26583/bit.2024.4.02.
8. Rastello F., Tichadou F. B. SSA-based Compiler Design. – Springer, 2022. – ISBN 978-3-030-80514-2. DOI: https://doi.org/10.1007/978-3-030-80515-9.
9. Khedler U. GCC Translation Sequence and Gimple IR. URL: https://reup.dmcs.pl/wiki/images/d/da/Gcc-gimple.pdf, (дата обращения: 02.10.2024).
10. Min-Yih H. LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries. – Packt Publishing, 2021. – ISBN 978-1-83882-495-2.
11. Egorov V. Yu. Development of the QP OS operating system // New information technologies and systems: A collection of scientific articles based on the materials of the XVII International Scientific and Technical Conference, Penza, November 18-19, 2020. – Penza: Penza State University, 2020. – pp. 45– 47. – EDN EJXOIX.

8 Sergey K. Muravyev, Ph.D, Head of Department of NTP Cryptosoft, Penza, smurav@mail.ru