УДК 004.056
doi: 10.26583/bit.2024.4.02

Сергей К. Муравьёв
ООО НТП «Криптософт»,
ул. Лермонтова, 3, Пенза, 440026, Россия
e-mail: smurav@mail.ru, https://orcid.org/0009-0008-2013-3456

УГРОЗЫ БЕЗОПАСНОСТИ ИНФОРМАЦИИ СО СТОРОНЫ МОДУЛЕЙ РАСШИРЕНИЯ ДЛЯ ОПТИМИЗИРУЮЩИХ КОМПИЛЯТОРОВ

Аннотация. Предметом статьи являются модули расширения или плагины для оптимизирующих компиляторов, подключение которых осуществляется через штатные программные интерфейсы, предусмотренные для повышения функциональности таких компиляторов и эффективности разрабатываемого с их помощью программного обеспечения. Цель статьи заключается в анализе угроз безопасности информации при разработке программного обеспечения, связанных с возможностью создания и использования злоумышленниками подобных модулей расширения, а также в формировании рекомендаций, которые способствуют нейтрализации рассмотренных угроз. Актуальность работы определяется действующими нормативно-техническими требованиями к разработчикам безопасного программного обеспечения по проведению анализа угроз безопасности информации со стороны средств разработки программного обеспечения, одним из ключевым элементом которых являются оптимизирующие компиляторы. В статье приведён краткий обзор существующих нормативно-технических требований к средам разработки безопасного программного обеспечения. Рассмотрены особенности анализа и трансформации исходного кода современными оптимизирующими компиляторами в процессе оптимизации исходного кода. На примере среды разработки LLVM в среде операционной системы Linux показана возможность практической реализации угроз безопасности информации со стороны модуля расширения для оптимизирующего компилятора, который меняет конвейер оптимизации таким образом, что алгоритм функционирования целевого приложения принципиально меняется. В итоге проанализированы сложности выявления подобных угроз, а также даны рекомендации по их нейтрализации. Представленные в статье сведения могут быть использованы при проектировании и реализации средств разработки безопасного программного обеспечения (РБПО), а также при внедрении соответствующих процессов РБПО.

Ключевые слова: Средства разработки, программное обеспечение, компилятор, модуль расширения, плагин, угроза безопасности информации, РБПО.
Для цитирования: МУРАВЬЁВ, Сергей К. УГРОЗЫ БЕЗОПАСНОСТИ ИНФОРМАЦИИ СО СТОРОНЫ МОДУЛЕЙ РАСШИРЕНИЯ ДЛЯ ОПТИМИЗИРУЮЩИХ КОМПИЛЯТОРОВ. Безопасность информационных технологий, [S.l.], т. 31, № 4, с. 44–55, 2024. ISSN 2074-7136. URL: https://bit.spels.ru/index.php/bit/article/view/171. DOI: http://dx.doi.org/10.26583/bit.2024.4.02.

Sergey K. Muravyev
NTP Cryptosoft LLC,
3 Lermontova str., Penza, 440026, Russia
e-mail: smurav@mail.ru, https://orcid.org/0009-0008-2013-3456

Information security threats of optimizing compilers' plugins

Abstract. The subject of the article is plugins for optimizing compilers, which are connected through standard software interfaces provided to improve the functionality of such compilers and the efficiency of the software developed with their help. The purpose of the article is to analyze information security threats in software development related to the possibility of creating and using such extension modules by hackers, as well as to form recommendations that help neutralize the threats considered. 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, the key element of which are optimizing compilers. The article provides a brief overview of the existing regulatory and technical requirements for secure software development environments. The features of the analysis and transformation of the source code by modern optimizing compilers in the process of optimizing the source code are considered. Using the example of the LLVM development environment in the Linux operating system environment, the possibility of practical implementation of information security threats from the extension module for the optimizing compiler, which changes the optimization pipeline in such a way that the algorithm of the target application functioning fundamentally changes, is shown. As a result, the difficulties of identifying such threats are also analyzed, as well as recommendations for their neutralization are given. The information presented in the article can be used in the design and implementation of secure software development tools, as well as in the implementation of appropriate processes.
Keywords: Development tools, software, extension module, plug-in, information security threats.
For citation: MURAVYEV, Sergey 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. URL: https://bit.spels.ru/index.php/bit/article/view/171. DOI: http://dx.doi.org/10.26583/bit.2024.4.02.

Введение

В настоящее время значительное внимание уделяется вопросам разработки безопасного программного обеспечения (ПО). При этом выделяется вопрос снижения угроз безопасности информации при разработке ПО со стороны инструментальных средств. Связано это с тем, что уязвимости могут появляться в программе в результате оптимизаций кода, выполняемых компилятором[1], а в качестве источников угроз безопасности информации при разработке ПО могут выступать инструментальные средства, применяемые при разработке ПО, алгоритм работы которых может стать причиной внедрения в ПО уязвимостей[2].

Современные инструментальные средства, используемые при разработке ПО, характеризуются гибкостью в настройке и широкими возможностями по расширению функционала, а алгоритмы их работы предполагают возможность разработки и подключения различных модулей расширения для достижения этих свойств. Ключевым элементом большинства современных средств разработки ПО являются оптимизирующие компиляторы, которые также предоставляют штатные программные интерфейсы для разработки и подключения расширений, способных изменить процесс оптимизации исходного кода. Таким образом, очевидна актуальность исследования угроз безопасности информации со стороны модулей расширения для оптимизирующих компиляторов, которым и посвящено данное исследование.

1. Моделирование атак на конвейер оптимизации

Наиболее известные компиляторы, входящие в состав GCC[3] и LLVM[4], в процессе компиляции применяют к исходному коду множество различных операций для его анализа и трансформации, называемые проходами. Проходы оптимизации объединяются в упорядоченные цепочки или конвейеры, управляемые менеджером проходов. При этом компиляторы предоставляют разработчикам несколько типовых конвейеров оптимизации,

[1]ГОСТ Р 71206 – 2024. Защита информации. Разработка безопасного программного обеспечения. Безопасный компилятор языков С/С++. Общие требования.
[2]ГОСТ Р 58412 – 2019. Защита информации. Разработка безопасного программного обеспечения. Угрозы безопасности информации при разработке программного обеспечения.
[3]GCC, the GNU Compiler Collection. URL: https://gcc.gnu.org/ (дата обращения: 19.09.2024).
[4]The LLVM Compiler Infrastructure. URL: https://llvm.org/ (дата обращения: 19.09.2024).

которые отвечают за общий уровень оптимизации целевого приложения, его размер, или скорость исполнения. А также компиляторы предоставляют разработчикам штатные программные интерфейсы для разработки и подключения собственных проходов с целью изменения и расширения типовых конвейеров оптимизации [1]. В зависимости от типа и версии компилятора эти типовые конвейеры оптимизации могут содержать несколько сотен различных проходов. На рис. 1 показан состав типового конвейера для второго уровня оптимизации компилятора из состава LLVM.

```
opt srand.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. 1. Состав типового конвейера для второго уровня оптимизации LLVM

Очевидно, что подавляющее большинство разработчиков, повседневно применяющих данный или подобный конвейер, даже не представляют назначение каждой операции и не проверяют состав перечня проходов на предмет корректности или безопасности для целевого приложения.

Для анализа возможностей потенциального злоумышленника, которые открывают данные особенности современных оптимизирующих компиляторов, рассмотрим простейшее тестовое приложение на языке C++ [2], которое формирует несколько числовых последовательностей с помощью генератора случайных чисел. Предполагается, что каждая формируемая последовательность будет уникальна за счёт инициализации генератора случайных чисел различными значениями. Исходный код приложения показан на рис. 2, а результат исполнения скомпилированного исполняемого файла – на рис. 3. Выбор rand [3] в качестве генератора случайных чисел сделан для упрощения объяснений представленных примеров, хотя в настоящее время рекомендуется использовать более современные альтернативы [4]. Общность выводов при этом никак не снижается. Рассмотренные далее методы атаки могут быть с успехом применены к любым другим генераторам случайных чисел.

srand.cpp

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4 int main() {
5   for(int i{0}; i < 10; ++i) {
6     unsigned int seed = i + 1;
7     srand(seed);
8     for(int j{0}; j < 10; ++j) {
9       cout << rand() % 10 << " ";
10    }
11    cout << endl;
12  }
13  return 0;
14 }
```

Рис. 2. Исходный код тестового приложения

```
clang srand.cpp -o srand ...
./srand
    3675356291
    0985184757
    6580502614
    1346375385
    5502620931
    1525856328
    7991536703
    6429132173
    5457030822
    5885875177
```

Рис. 3. Результат исполнения тестового приложения

Рассмотрим возможные атаки злоумышленника на тестовое приложение, которые позволят раскрыть информацию о фактических числовых последовательностях, обрабатываемых тестовым приложением.

Для начала рассмотрим атаку, связанную с нарушением конфиденциальности исходного кода приложения. Получив доступ к исходному коду, злоумышленник сможет его проанализировать и предсказать используемые значения для инициализации генератора случайных чисел, а, значит, и результирующие последовательности. В этом случае злоумышленник может достичь своих целей без дополнительных действий. Данная атака особенно актуальна, если предположить, что тестовое приложение относится к программному обеспечению с открытым исходным кодом.

Нарушитель также может попытаться нарушить целостность исходного кода и внедрить в него вредоносные правки непосредственно в исходный код приложения и добиться нужного поведения программы. Также злоумышленник может попытаться внедрить вредоносные правки в компилятор или другие инструментальные средства для получения исполняемых файлов с нужным поведением.

В силу того, что данные виды атак относительно просто выявить и противодействовать им, рассматривать более подробно их не будем и сосредоточимся на атаке, которая не связана с нарушением конфиденциальности и целостности исходного

кода, а также целостности исполняемых файлов компилятора и других инструментальных средств разработки.

Рассмотрим подробнее атаку злоумышленника, связанную с использованием штатных интерфейсов компилятора для встраивания вредоносных проходов оптимизации, которые способны изменить исходный алгоритм программы. На начальных этапах работы компилятор преобразует исходный код в промежуточный код [5], к которому применяются различные преобразования и далее на его основе осуществляется генерация целевого кода. Далее, без ограничения общности, будем рассматривать оптимизирующий компилятор языка C++ из состава экосистемы LLVM 20 в среде операционной системы Linux [6].

В промежуточном представлении LLVM [7] исходный код разбивается на базовые блоки, содержащие простые последовательности инструкций без ветвлений. Имена функций в промежуточном представлении начинаются с символа @, а имена переменных заменяются на числа, перед которыми ставится символ %.

В представленном на рис. 4 базовом блоке можно увидеть операции увеличения значения переменной %3 на единицу и последующую передачу получившегося значения переменной %4 в функцию @srand, которые соответствуют строкам 6 и 7 исходной программы на языке C++.

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

```
; ModuleID = 'srand.cpp'
...
target triple = "x86_64-unknown-linux-gnu"
...
2:                                ; preds = %0, %25
  %3 = phi i32 [ 0, %0 ], [ %4, %25 ]
  %4 = add nuw nsw i32 %3, 1
  tail call void @srand(i32 noundef %4) #6
  br label %30
...
```

```
...
5    for(int i{0}; i < 10; ++i) {
6      unsigned int seed = i + 1;
7      srand(seed);
...
```

Рис. 4. Промежуточное представление исходного кода тестового приложения

Используя штатные программные интерфейсы компилятора, злоумышленник может разработать вредоносный проход оптимизации, который осуществляет перебор базовых блоков и входящих в них инструкций. При обнаружении инструкции для вызова функции srand осуществляется замена любого входного значения, предусмотренного алгоритмом работы программы, на целочисленную константу, известную злоумышленнику. При этом злоумышленнику нет необходимости иметь прямой доступ к исходному коду приложения. Вредоносный проход может выявить и изменить вызов любой системной функции и таким образом данный метод атаки может быть с успехом применён к любым другим генераторам случайных чисел, в том числе и аппаратным. На рис. 5 показана часть исходного кода вредоносного прохода оптимизации, реализующего данный алгоритм.

srandpass.cpp

```
31 for (auto &BB : F) {
32   for (auto &I : BB) {
33     CallInst *CI = dyn_cast<CallInst>(&I);
34     if (nullptr == CI) continue;
35     Function *CF = CI->getCalledFunction();
36     if (nullptr == CF) continue;
37     if (0 == CF->getName().compare("srand")) {
38       Value *V = CI->getOperand(0);
39       Value *NV = ConstantInt::get(V->getType(), 1234567, false);
40       CI->setArgOperand(0, NV);
41           return PreservedAnalyses::none();
42     }
43   }
44 }
```

Рис. 5. Ключевой участок исходного кода вредоносного прохода

После проведения вредоносной оптимизации алгоритм работы исходного приложения принципиально меняется при сохранении целостности исходного кода. На рис. 6 показано изменённое промежуточное представление исходного кода тестового приложения, полученное после выполнения вредоносного прохода оптимизации. На рис. 7 приведен результат исполнения скомпилированного тестового приложения, после вредоносной оптимизации, на котором видно, что тестовое приложение формирует уже неслучайные числовые последовательности, каждый элемент которых известен злоумышленнику.

```
clang -fpass-plugin=./srandpass.so -S -emit-llvm -c srand.ll -o srand_instr.ll
cat ./srand_instr.ll
```

```
...
2:                                                ; preds = %25, %0
  %3 = phi i32 [ 0, %0 ], [ %4, %25 ]
  %4 = add nuw nsw i32 %3, 1
  tail call void @srand(i32 noundef 1234567) #6
  br label %30
...
```

```
...
5    for(int i{0}; i < 10; ++i) {
6      unsigned int seed = i + 1;
7      srand(1234567);
...
```

Рис. 6. Промежуточное представление исходного кода после вредоносной оптимизации

```
clang srand_instr.ll -o srand ...
clang srand.cpp -o srand -fpass-plugin=./srandpass.so ...
./srand
```

```
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
```

Рис. 7. Результат исполнения тестового приложения после вредоносной оптимизации

2. Сложности обнаружения атак на конвейер оптимизации

Основные сложности обнаружения подобных вмешательств в конвейер оптимизации заключаются в том, что вывод современных инструментальных средств настолько многословен и такую информацию сложно считать человекочитаемой. Без использования дополнительных средств автоматизации обнаружить вредоносные вмешательства в конвейер оптимизации довольно сложно. В этом легко убедиться, если взглянуть на рис. 8, где показан изменённый типовой конвейер для второго уровня оптимизации, показанный ранее на рис. 1.

```
opt srand.ll -load-pass-plugin=./srandpass.so -passes="default<O2>" -S -o srand_instr.ll -print-pipeline-passes
```

```
annotation2metadata,forceattrs,srandpass,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. 8. Состав конвейера для второго уровня оптимизации LLVM с вредоносным проходом

Сложности обнаружения подобных атак также связаны с тем, что разработчики, как правило, взаимодействуют не с компилятором, а с его драйвером, который уже формирует

полный набор параметров компилятора. Приведенный на рис. 8 и ранее вызов clang на самом деле представляет собой вызов драйвера для компилятора clang-20, который уже выполняет компиляцию. Драйвер clang реализует интерфейс командной строки, аналогичный компилятору gcc. Также в состав LLVM входит драйвер clang-cl [8], который реализует интерфейс командной строки, аналогичный основному компилятору для операционной системы Windows – cl.exe[5].

Какой бы драйвер ни выбрал разработчик, фактический перечень параметров основного компилятора будет слишком многословным, чтобы его можно было считать человекочитаемым, как показано на рис. 9.

```
clang srand.cpp -o srand -fpass-plugin=./srandpass.so -### ...
```

```
"/home/dev/llvm20/bin/clang-20" "-cc1" "-triple" "x86_64-unknown-linux-gnu" "-emit-obj" "-dumpdir" "srand-" "-disable-free" "-clear-ast-before-backend" "-disable-llvm-verifier" "-discard-value-names" "-main-file-name" "srand.cpp" "-mrelocation-model" "pic" "-pic-level" "2" "-pic-is-pie" "-mframe-pointer=all" "-fmath-errno" "-ffp-contract=on" "-fno-rounding-math" "-mconstructor-aliases" "-funwind-tables=2" "-target-cpu" "x86-64" "-tune-cpu" "generic" "-debugger-tuning=gdb" "-fdebug-compilation-dir=/home/dev/git/cg/src/srand" "-fcoverage-compilation-dir=/home/dev/git/cg/src/srand" "-resource-dir" "/home/dev/llvm20/lib/clang/20" "-I" "/usr/include/c++/11/" "-I" "/usr/include/x86_64-linux-gnu/c++/11/" "-internal-isystem" "/usr/lib/gcc/x86_64-linux-gnu/12/../../../../include/c++" "-internal-isystem" "/usr/lib/gcc/x86_64-linux-gnu/12/../../../../include/c++/x86_64-linux-gnu" "-internal-isystem" "/usr/lib/gcc/x86_64-linux-gnu/12/../../../../include/c++/backward" "-internal-isystem" "/home/dev/llvm20/lib/clang/20/include" "-internal-isystem" "/usr/local/include" "-internal-isystem" "/usr/lib/gcc/x86_64-linux-gnu/12/../../../../x86_64-linux-gnu/include" "-internal-externc-isystem" "/usr/include/x86_64-linux-gnu" "-internal-externc-isystem" "/include" "-internal-externc-isystem" "/usr/include" "-fdeprecated-macro" "-ferror-limit" "19" "-fgnuc-version=4.2.1" "-fskip-odr-check-in-gmf" "-fcxx-exceptions" "-fexceptions" "-fcolor-diagnostics" "-fpass-plugin=./srandpass.so" "-faddrsig" "-D__GCC_HAVE_DWARF2_CFI_ASM=1" "-o" "/tmp/srand-0028d5.o" "-x" "c++" "srand.cpp""/usr/bin/ld" "--hash-style=gnu" "--eh-frame-hdr" "-m" "elf_x86_64" "-pie" "-dynamic-linker" "/lib64/ld-linux-x86-64.so.2" "-o" "srand" "/lib/x86_64-linux-gnu/Scrt1.o" "/lib/x86_64-linux-gnu/crti.o" "/usr/lib/gcc/x86_64-linux-gnu/12/crtbeginS.o" "-L/usr/lib/gcc/x86_64-linux-gnu/11" "-L/usr/lib/gcc/x86_64-linux-gnu/12" "-L/usr/lib/gcc/x86_64-linux-gnu/12/../../../../lib64" "-L/lib/x86_64-linux-gnu" "-L/lib/../lib64" "-L/usr/lib/x86_64-linux-gnu" "-L/usr/lib/../lib64" "-L/lib" "-L/usr/lib" "/tmp/srand-0028d5.o" "-lstdc++" "-lgcc" "--as-needed" "-lgcc_s" "--no-as-needed" "-lc" "-lgcc" "--as-needed" "-lgcc_s" "--no-as-needed" "/usr/lib/gcc/x86_64-linux-gnu/12/crtendS.o" "/lib/x86_64-linux-gnu/crtn.o"
```

Рис. 9. Фактический перечень параметров компилятора для сборки тестового приложения

Сложности обнаружения вмешательств в конвейер оптимизации также могут быть связаны с тем, что ряд параметров компилятора могут определяться неявно. Такое неявное определение возможно, например, за счёт изменения различных переменных окружения операционной системы. Так переменная окружения LD_LIBRARY_PATH [9] операционной системы Linux влияет на порядок поиска подключаемых библиотек. Если разработчик ПО использует дополнительные подгружаемые расширения конвейера оптимизации и указывает относительные пути к соответствующим библиотекам, то их поиск будет осуществляться в соответствии с системными настройками окружения операционной системы. В этом случае злоумышленник может попытаться подменить легитимный проход вредоносным, если изменит порядок поиска подключаемых библиотек на уровне операционной системы.

На рис. 10 показан пример компиляции с использованием легитимного прохода, при котором получаемый исполняемый файл тестового приложения работает корректно.

[5]Use the Microsoft C++ toolset from the command line. URL: https://learn.microsoft.com/en-us/cpp/build/building-on-the-command-line (дата обращения: 19.09.2024)

На рис. 11 показан вывод атакованного тестового приложения, при компиляции которого выполнена подмена подключаемой библиотеки, отвечающей за реализацию прохода оптимизации goodpass. В обоих примерах параметры вызова компилятора идентичны. Меняется лишь значение переменной среды окружения LD_LIBRARY_PATH.

```
export LD_LIBRARY_PATH="/good/path/;..."
clang srand.cpp -o srand -fpass-plugin=goodpass.so ...
./srand
```

```
3675356291
0985184757
6580502614
1346375385
5502620931
1525856328
7991536703
6429132173
5457030822
5885875177
```

Рис. 10. Пример компиляции с использованием легитимного прохода оптимизации

```
export LD_LIBRARY_PATH="/wrong/path/;..."
clang srand.cpp -o srand -fpass-plugin=goodpass.so ...
./srand
```

```
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
```

Рис. 11. Пример компиляции с использованием вредоносного прохода оптимизации

Также необходимо отметить, что компиляторы редко применяются в отрыве от таких инструментальных средств разработки, как интегрированные среды разработки или генераторы сборочных сценариев. Данные средства автоматизируют подготовку целевых сценариев сборки, которые уже активируют процессы компиляции исполняемых файлов. При этом ручной контроль таких сценариев крайне сложен, а существующие средства статического и динамического анализа не обеспечивают контроль их безопасности [10].

Использование генераторов сборочных сценариев открывает ещё больше возможностей для неявного определения параметров сборки и приводит к дальнейшему увеличению объёма выводимой информации. На рис. 12 и 13 показаны результаты корректной и вредоносной сборок тестового приложения, соответственно, с помощью популярного генератора сборочных сценариев CMake [11] при которых осуществляется неявное определение флагов компиляции с помощью переменной окружения CXXFLAGS. Полный вывод, который отображается разработчику в процессе сборки, сложно уместить на рисунках, но стоит отметить, что он абсолютно идентичен в обоих случаях. Разницу

можно обнаружить лишь при детальном изучении содержания сгенерированных сценариев сборки, но в ручном режиме это практически неосуществимо.

```
cmake -G Ninja –B build
cmake  --build build
./build/srand
```

```
3675356291
0985184757
6580502614
1346375385
5502620931
1525856328
7991536703
6429132173
5457030822
5885875177
```

Рис. 12. Корректная сборка тестового приложения с помощью CMake

```
CXXFLAGS="-fpass-plugin=srandpass.so " cmake -G Ninja –B build
cmake  --build build
./build/srand
```

```
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
4290083546
```

Рис. 13. Вредоносная сборка тестового приложения с помощью CMake

3. Противодействие атакам на конвейер оптимизации

Для снижения подобных угроз со стороны подключаемых модулей для оптимизирующих компиляторов в первую очередь необходимо зафиксировать перечень разрешенных к применению проходов оптимизации и реализовать общие требования к безопасному компилятору языков C/C++ в части формирования и контроля базы данных компиляции.

Так как существующие средства статического анализа [12] и динамического анализа [13] ограничиваются традиционными языками программирования и не обеспечивают контроль языков, используемых генераторами сценариев сборки, то потребуется разработка и внедрение новых средств анализа.

В обязательном порядке необходимо осуществлять контроль системного окружения операционной системы, в рамках которого ведётся разработка. При сборке компилятора необходимо запретить динамически подключаемые расширения или реализовать контроль их аутентичности.

А в качестве самого надёжного решения можно порекомендовать использовать защищённую операционную систему QP ОС [14] в качестве платформы для разработки

безопасного программного обеспечения. Средства обеспечения замкнутой программной среды в QP ОС [15] обеспечивают полную защиту об описанных в статье угроз.

Заключение

В статье показана возможность реализации угроз безопасности информации со стороны модулей расширения для оптимизирующих компиляторов на примере дополнительного элемента конвейера оптимизации, способного принципиально изменить алгоритм работы ПО и существенно снизить безопасность информации, обрабатываемой таким программным обеспечением. При этом конвейер оптимизации менялся с помощью штатных интерфейсов компилятора, а целостность исходного кода ПО и инструментальных средств, применяемых при разработке ПО, не нарушалась. Также проанализированы сложности выявления подобных угроз и даны рекомендации по их нейтрализации.

Представленные в статье сведения могут быть использованы при проектировании и реализации средств разработки безопасного программного обеспечения, а также при внедрении соответствующих процессов разработки безопасного программного обеспечения[6].

СПИСОК ЛИТЕРАТУРЫ:

1. 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.
2. Страуструп Б. Язык программирования C++, 3-е изд.: Пер. с англ. М.: «Издательство БИНОМ», 1999. – 991 с. ISBN 5-7989-0127-0.
3. Белов А.А., Калиткин Н.Н., Тинтул М.А. Ненадежность известных генераторов псевдослучайных чисел. Журнал вычислительной математики и математической физики. 2020, т. 60, № 11, с. 1807–1814. DOI: 10.31857/S0044466920110046. – EDN: CTJCWS.
4. Дейтел П., Дейтел Х. C++20 для программистов. СПб.: Питер, 2024. – 1056 с. – ISBN 978-5-4461-2359-9.
5. Ахо А., Лам М., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. М.: ООО «И.Д. Вильямс», 2018. – 1184 с. ISBN 978-5-8459-1932-8.
6. Негус К. Библия Linux. 10-е издание. СПб.: Питер, 2022. – 928 с. ISBN 978-5-4461-1797-0.
7. Min-Yih H. LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries. Packt Publishing, 2021. ISBN 978-1-83882-495-2.
8. Wennborg H. clang-cl. What it is, how it works, and how to use it. URL: https://learn.microsoft.com/en-us/cpp/build/building-on-the-command-line (дата обращения: 19.09.2024).
9. Barr D. Why LD_LIBRARY_PATH is bad. URL: http://xahlee.info/UnixResource_dir/_/ldpath.html (дата обращения: 19.09.2024).
10. Kudrjavets G., Kumar A., Thomas J. and Rastogi A. The Devil Is in the Command Line: Associating the Compiler Flags With the Binary and Build Metadata. IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Lisbon, Portugal. 2024, p. 134–136. DOI: 10.1145/3639477.3639737.
11. Berner D., Gilor M. K. CMake Best Practices: Discover Proven Techniques for Creating and Maintaining Programming Projects with CMake. Packt Publishing, 2022. ISBN 978-1803244242.
12. Муравьёв С.К., Бальзаникова Е. А., Орлов М.А. Обзор возможностей статических анализаторов исходного кода, предоставляемых средствами компиляции программного обеспечения. Состояние и перспективы развития современной науки по направлению «Информационные технологии в Вооруженных Силах Российской Федерации»: Сборник III Всероссийской научно-технической конференции, Анапа, 21–22 марта 2024 года. Анапа: Военный инновационный технополис «ЭРА». 2024, с. 82–101. – EDN: CGTPNV.

[6]ГОСТ Р 56939 – 2016. Защита информации. Разработка безопасного программного обеспечения. Общие требования.

13. Вьюкова Н.И., Галатенко В.А., Самборский С.В. Средства динамического анализа программ в компиляторах GCC и CLANG. Программирование. 2020, № 4, с. 46–64. DOI: 10.31857/S0132347420010082. – EDN: NTIQRW.
14. Егоров В.Ю. Развитие операционной системы QP ОС. Новые информационные технологии и системы: Сборник научных статей по материалам XVII Международной научно-технической, Пенза, 18–19 ноября 2020 года. Пенза: Пензенский государственный университет. 2020, с. 45–47. – EDN: EJXOIX.
15. Егоров В.Ю., Гуреев В.О. Администрирование защищённой операционной системы QP ОС. Пенза, 2024. – 552 с. ISBN 978-5-6040973-4-2.

REFERENCES:

[1] 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.
[2] Stroustrup B. The C++ Programming Language, Third Edition. AT&T, 1997. ISBN 0-201-88954-4.
[3] Belov A.A., Tintul M.A., Kalitkin N.N. Unreliability of Available Pseudorandom Number Generators. Computational Mathematics and Mathematical Physics. 2020, v. 60, no. 11, p. 1747–1753. DOI: 10.1134/S0965542520110044. – EDN: RIJGWM.
[4] Deitel P., Deitel H. C++20 for Programmers. Pearson, 2022. ISBN 978-0136905691.
[5] Aho A., Lam M., Sethi R., Ullman J. Compilers. Principles, Techniques, & Tools, Second Edition. – Addison Wesley, 2007. ISBN 0-321-48681-1.
[6] Negus C. Linux Bible, Tenth Edition. Wiley, 2020. ISBN 978-1119578888.
[7] Min-Yih H. LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries. Packt Publishing, 2021. ISBN 978-1-83882-495-2.
[8] Wennborg H. clang-cl. What it is, how it works, and how to use it. URL: https://learn.microsoft.com/en-us/cpp/build/building-on-the-command-line (accessed: 19.09.2024).
[9] Barr D. Why LD_LIBRARY_PATH is bad. URL: http://xahlee.info/UnixResource_dir/_/ldpath.html (accessed: 19.09.2024).
[10] Kudrjavets G., Kumar A., Thomas J. and Rastogi A. The Devil Is in the Command Line: Associating the Compiler Flags With the Binary and Build Metadata. IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Lisbon, Portugal. 2024, p. 134–136. DOI: 10.1145/3639477.3639737.
[11] Berner D., Gilor M. K. CMake Best Practices: Discover Proven Techniques for Creating and Maintaining Programming Projects with CMake. Packt Publishing, 2022. ISBN 978-1803244242.
[12] Muravyov S.K., Balzanikova E. A., Orlov M.A. Overview of the capabilities of static source code analyzers provided by software compilation tools. Information technologies in the Armed Forces of the Russian Federation: Collection of the III All-Russian Scientific and Technical Conference, Anapa, March 21–22, 2024. Anapa: Military innovative technopolis "ERA". 2024, c. 82–101 (in Russian) – EDN: CGTPNV.
[13] V'yukova N.I., Galatenko V.A., Samborskii S.V. Dynamic Program Analysis Tools in GCC and CLANG Compilers. Programming and Computer Software. 2020, v. 46, p. 281–296. DOI: 10.1134/S0361768820010089. – EDN: TGZZDK.
[14] 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, p. 45–47 (in Russian). – EDN: EJXOIX.
[15] Egorov V.Yu., Gureev V.O. Administration of the QP OS secure operating system. Penza, ISBN 978-5-6040973-4-2. 2024. – 552 p. (in Russian).

Поступила в редакцию – 15 августа 2024 г. Окончательный вариант – 30 сентября 2024 г.
Received – August 15, 2024. The final version – September 30, 2024.