Hardening-флаги в НАЙС.ОС: как мы защищаем вас от эксплойтов

В НАЙС.ОС по умолчанию включены ключевые hardening-флаги, которые значительно повышают устойчивость системы к эксплуатации уязвимостей. Эти флаги активируют защитные механизмы на уровне бинарных файлов: PIE (перемещаемые исполняемые файлы), RELRO (защита таблиц GOT), SSP (Stack Smashing Protection), ASLR (рандомизация адресного пространства), FORTIFY_SOURCE (дополнительные проверки функций libc). Вместе они делают практически невозможным эксплуатацию типовых багов вроде переполнения буфера, возврата в libc или ROP-атак без сложных обходов. Эти защиты активированы как для системных сервисов, так и для пользовательских приложений, включая sudo, sshd, rpm, gpg. Пользователи получают безопасную по умолчанию систему, а разработчики — возможность собирать устойчивые к атакам приложения без лишних усилий.

1. Зачем нужны Hardening-флаги

Современные атаки на приложения редко опираются только на логические ошибки. Чаще всего они используют **низкоуровневые уязвимости** — такие как переполнение буфера, повреждение указателей, использование уже освобождённой памяти (use-after-free), переходы на произвольный код (return-to-libc, ROP).

Даже если у вас нет критических багов в коде, типовая уязвимость в сторонней библиотеке может быть использована для компрометации всей системы. Здесь на помощь приходят *hardening*-флаги — специальные параметры компиляции и линковки, которые:

- предотвращают исполнение кода из стека или кучи (NX, RELRO);

- делают невозможным предсказание адресов функций (ASLR, PIE);
- заставляют программы падать безопасно при попытке переписать стек (SSP);
- вставляют дополнительные проверки при работе со строками (FORTIFY_SOURCE).

Во многих случаях **hardening мешает эксплойту даже при наличии уязвимости** — либо усложняя атаку до невозможности, либо вызывая безопасное завершение процесса до компрометации.

🧱 Исторические примеры

- **Heartbleed (2014)** — утечка памяти из-за отсутствия проверки границ;
- **Dirty COW (2016)** — гонка в доступе к памяти;
- **Stack Clash (2017)** — переполнение стека с перезаписью heap;
- **Return-to-libc / ROP** — типовые техники обфускации исполнения.

Современные флаги компиляции блокируют или смягчают большинство таких атак.

Итог: hardening — это дешёвая, эффективная и прозрачная форма защиты, которая работает на уровне компиляции и распространяется на всё системное и пользовательское ПО.

2. Какие механизмы мы используем в НАЙС.ОС

В **НАЙС.ОС** используется полный набор актуальных защитных флагов компиляции, линковки и исполнения. Они применяются ко всем системным пакетам по умолчанию — как при сборке в RPM-спеке, так и через глобальные политики сборки.

Механизм	Описание	Флаг / настройка
PIE (Position Independent Executables)	Позволяет размещать исполняемый код в случайном месте памяти (требуется для ASLR).	`-fPIE -pie`
RELRO (Relocation Read-Only)	Делает GOT-таблицы только для чтения, предотвращает подмену функций.	`-Wl,-z,relro -Wl,-z,now`
ASLR (Address Space Layout Randomization)	Случайное размещение стека, heap, библиотек и исполняемых файлов.	Активируется в ядре и через `PIE`

Механизм	Описание	Флаг / настройка
SSP (Stack Smashing Protection)	Вставляет канарейку (canary) в стек, проверяет перед возвратом из функции.	`-fstack-protector-strong`
FORTIFY_SOURCE	Расширенные проверки стандартных функций (strcpy, memcpy и др.) с размером буфера на этапе компиляции и исполнения.	`-D_FORTIFY_SOURCE=2` (с `-O2` и выше)

Эти флаги применяются через `%optflags` в RPM-спецификациях и встроены в макросы `gcc` / `clang` на этапе сборки. Также используется проверка `hardening-check` и Lint-сценарии в CI.

🎁 **Пример:** Утилита `sudo` в НАЙС.ОС собирается с полным набором: PIE + RELRO + SSP + FORTIFY. Попытка переполнить буфер вызывает защитный краш, а не выполнение shell-кода.

3. Как проверить, что защита работает

Проверить наличие защитных механизмов можно как на уровне исполняемых файлов, так и на уровне поведения системы. Ниже приведены основные способы.

🔍 Проверка флагов бинарника

Для анализа включённых защит можно использовать утилиту `checksec`:

```
checksec --file /usr/bin/sudo
```

Пример вывода:

```
RELRO           Full RELRO
Stack Canary    Yes
NX              Enabled
PIE             Enabled
FORTIFY         Enabled
```

Также можно использовать встроенную команду `readelf`:

- `readelf -h /usr/bin/ssh` — покажет, PIE или нет (ET_DYN = PIE);
- `readelf -l /usr/bin/bash | grep GNU_RELRO` — наличие секции RELRO;
- `strings /usr/bin/bash | grep __stack_chk` — наличие защиты SSP;
- `objdump -d /usr/bin/ls | grep strcpy` — поможет найти потенциально опасные функции;

Тест поведения (stack smashing)

Можно скомпилировать простой уязвимый код и проверить, сработает ли защита:

```
#include <string.h>

int main() {
char buf[8];
strcpy(buf, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
return 0;
}
```

Скомпилируйте с SSP:

```
gcc -fstack-protector-strong test.c -o test
```

При запуске программа должна завершиться с ошибкой:

```
*** stack smashing detected ***: terminated
```

↩ Проверка ASLR

Убедитесь, что ASLR включён в ядре:

```
cat /proc/sys/kernel/randomize_va_space
```

Значения:

- `0` — отключено;
- `1` — частично включено (stack, mmap);
- `2` — полностью включено (PIE + libs + stack + heap).

Важно: в НАЙС.ОС по умолчанию `2`, и все пользовательские процессы собираются с

`PIE`.

4. Чем отличается от других дистрибутивов

Многие современные дистрибутивы Linux включают частичную поддержку hardening-флагов — но не все делают это системно и по умолчанию для **всех** пакетов. **НАЙС.ОС** отличается тем, что:

- 🔒 **Все системные утилиты** и сервисы собираются с PIE, RELRO, SSP, FORTIFY и NX;
- ⚙️ **Политики безопасности применяются на уровне toolchain** — через глобальные `optflags`, `rpm macros` и компиляторы;
- 🔁 **ASLR включён по умолчанию** во всех профилях, включая headless-серверы и контейнеры;
- **CI и QA-пайплайны** включают автоматическую проверку на наличие всех защит в собранных бинарниках;
- **Утилита `hardening-check`** встроена в инструменты разработчика;
- 🔐 **Особое внимание уделяется критическим компонентам**: `sudo`, `sshd`, `gnupg`, `rpm`, `openssl` и другим.

📊 Таблица сравнения с популярными дистрибутивами

Дистрибутив	PIE	RELRO	SSP	FORTIFY	ASLR
НАЙС.ОС	✅ по умолчанию	✅ full (NOW)	✅ strong	✅ level 2	✅ включено
Ubuntu 22.04	⚠️ частично	✅	✅	✅	✅
Debian 12	⚠️ частично	⚠️ partial	✅	✅	✅
AlmaLinux / RHEL 9	✅ (в некоторых пакетах)	✅	✅	⚠️ не везде	✅

Таким образом, **НАЙС.ОС** стремится к максимально жёсткой, но прозрачной политике hardening-защиты — без необходимости ручной настройки со стороны пользователя или сборки «особых» пакетов.

5. Как это влияет на безопасность реальных систем

Hardening-флаги — это не просто академическая «проверка чекбоксов». Они дают **реальное преимущество** в противостоянии атакам нулевого дня, ошибкам в сторонних библиотеках и целевым эксплойтам.

🧱 Практический эффект

- 🛡 **SSP (stack protector)** предотвращает использование стековых переполнений — программа аварийно завершится до выполнения shellcode.
- 🔐 **RELRO + PIE + ASLR** усложняют или делают невозможным return-to-libc и ROP-атаки, за счёт непредсказуемости адресов и защиты GOT.
- **FORTIFY_SOURCE** внедряет дополнительные runtime-проверки в функции работы со строками и памятью (например, `strcpy`, `memcpy`).
- ⛔ **NX (non-exec stack)** полностью блокирует исполнение shellcode, даже если злоумышленник получил контроль над стеком.

Типовой сценарий: уязвимость в библиотеке

Допустим, в системе установлена библиотека, уязвимая к переполнению буфера (например, JPEG-декодер). Без hardening, злоумышленник может подставить обработанный файл, запустить ROP и получить shell.

С включённым hardening в НАЙС.ОС:

- Атака будет выявлена на стадии выполнения (SSP вызывает `__stack_chk_fail()`);
- Стек не содержит исполняемых страниц (NX);
- GOT защищена (RELRO), и злоумышленник не может перехватить вызов `free()` или `system()`;
- Адреса функций и libc меняются при каждой загрузке (PIE + ASLR);
- Вероятность успешной эксплуатации падает практически до нуля.

Итог: даже в случае 0-day или ошибки в софте стороннего поставщика, вы получаете ещё один уровень защиты — невидимый, автоматический и работающий по умолчанию.

6. Как включить в своих проектах

Включение hardening-флагов в своих C/C++-проектах — это простая и эффективная мера безопасности. Ниже — основные подходы для популярных систем сборки и компиляторов.

🔧 Makefile

Добавьте флаги в переменные `CFLAGS` и `LDFLAGS`:

```
CFLAGS += -O2 -fstack-protector-strong -D_FORTIFY_SOURCE=2 -fPIE
LDFLAGS += -Wl,-z,relro,-z,now -pie
```

🔧 CMake

В `CMakeLists.txt`:

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O2 -fstack-protector-strong -D_FORTIFY_SOURCE=2 -fPIE")
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,-z,relro,-z,now -pie")
```

🔧 Meson

Добавьте в `meson.build`:

```
add_project_arguments('-D_FORTIFY_SOURCE=2', '-fstack-protector-strong', '-fPIE',
language : 'c')
add_project_link_arguments('-Wl,-z,relro', '-Wl,-z,now', '-pie',
language : 'c')
```

🔧 Autotools

Перед `./configure` задайте переменные окружения:

```
export CFLAGS="-O2 -fstack-protector-strong -D_FORTIFY_SOURCE=2 -fPIE"
export LDFLAGS="-Wl,-z,relro,-z,now -pie"
./configure
```

Советы по отладке

- При использовании `-pie` убедитесь, что ваш загрузчик и runtime поддерживают ASLR.
- Не все флаги совместимы с LTO (`-flto`) — проводите отдельные тесты сборки.
- В случае shared-библиотек используйте `-fPIC` вместо `-fPIE` .
- Учитывайте, что `-D_FORTIFY_SOURCE` работает только с `-O1` и выше.
- Для C++ не забудьте применить флаги и к `CXXFLAGS` .

Итог: даже минимальный набор флагов может значительно повысить безопасность вашего приложения — особенно если оно работает с не доверенными данными.

7. Подводные камни и ограничения

Несмотря на очевидные плюсы, использование hardening-флагов имеет ряд ограничений и нюансов, о которых важно знать при разработке и отладке.

⚠ Производительность и real-time

- **PIE (Position Independent Executable)** вносит дополнительную нагрузку на динамическое разрешение адресов . В типичных задачах разница незаметна, но в real-time системах (например, embedded, SCADA, VoIP) она может повлиять на задержки и детерминизм.
- Рекомендуется измерять производительность вручную в таких случаях и отключать PIE/ASLR точечно для критичных компонентов.

Требования к toolchain

Полноценная поддержка всех флагов требует современного компилятора и библиотек. В **НАЙС.ОС** всё это включено из коробки, но для других систем важно учесть:

- **GCC 9.3** — поддержка `-fstack-protector-strong` , PIE по умолчанию и FORTIFY;
- **Glibc 2.31** — расширенные проверки `FORTIFY_SOURCE` и защита malloc-метаданных;
- **binutils 2.35** — корректная линковка с `-z relro` , `-z now` , и PIE.

⚙️ Влияние на отладку

- PIE/ASLR могут затруднить отладку, т.к. адреса в памяти меняются при каждом запуске. В GDB это можно обойти командой:

```
set disable-randomization off
```

- Отладка производительности (через `perf`, `valgrind`) также усложняется, особенно если используется `fortify` и `stack protector`, т.к. они могут прерывать выполнение при ошибках.
- Системы сборки с LTO и отложенной линковкой требуют особого внимания: не все hardening-флаги дружат с оптимизирующими сборками.

Вывод: hardening — это не серебряная пуля, а инструмент. Его следует применять осознанно, с пониманием контекста и ограничений.

8. Проверка на соответствие: CI + аудит

Одного включения hardening-флагов недостаточно — важно **проверять, что они действительно применились** . Особенно в масштабных проектах и в рамках требований сертификации ИБ. Ниже приведены подходы к автоматической проверке и аудиту.

✅ Проверка на уровне CI/CD

Включите автоматические проверки в ваши CI-пайплайны (GitHub Actions, GitLab CI, Jenkins и др.). Пример задачи на GitHub Actions:

```
- name: Проверка hardening-флагов
run: |
checksec --file=./your_binary
hardening-check ./your_binary
shell: bash
```

Также можно встроить вызов утилит в собственные build-сценарии и Makefile:

```
check-hardening:
checksec --file=bin/myapp
hardening-check bin/myapp
```

🔍 Утилиты для аудита

- `checksec` — универсальный инструмент, показывает наличие PIE, RELRO, NX, SSP и др.
- `hardening-check` — утилита от Debian Security, быстро проверяет соответствие общим правилам.
- `lintian` (в Debian/Ubuntu) — выявляет отсутствие флагов в .deb-пакетах.
- `rpm --eval %__global_ldflags` — проверка применяемых флагов при сборке пакетов в RPM-окружении.

📋 Интеграция с аудитом и сертификацией

- Для ОС, подлежащих сертификации ФСТЭК/Минобороны, важно зафиксировать применение флагов в **отчётах по защищённости** и при прохождении испытаний.
- В НАЙС.ОС отчёты `checksec` включаются в состав артефактов сборки.
- Возможно применение внешнего аудита (3rd party), особенно для компонентов безопасности, библиотек и служб.

Итог: систематическая проверка — важная часть цепочки доверия. Автоматизация аудита в CI помогает не допустить случайных ошибок и упростить сертификацию решений.

8. Проверка на соответствие: CI + аудит

Одного включения hardening-флагов недостаточно — важно **проверять, что они действительно применились** . Особенно в масштабных проектах и в рамках требований сертификации ИБ. Ниже приведены подходы к автоматической проверке и аудиту.

✔ Проверка на уровне CI/CD

Включите автоматические проверки в ваши CI-пайплайны (GitHub Actions, GitLab CI, Jenkins и др.). Пример задачи на GitHub Actions:

```
- name: Проверка hardening-флагов
```

```
run: |
checksec --file=./your_binary
hardening-check ./your_binary
shell: bash
```

Также можно встроить вызов утилит в собственные build-сценарии и Makefile:

```
check-hardening:
checksec --file=bin/myapp
hardening-check bin/myapp
```

🔍 Утилиты для аудита

- **`checksec`** — универсальный инструмент, показывает наличие PIE, RELRO, NX, SSP и др.
- **`hardening-check`** — утилита от Debian Security, быстро проверяет соответствие общим правилам.
- **`lintian`** (в Debian/Ubuntu) — выявляет отсутствие флагов в .deb-пакетах.
- **`rpm --eval %__global_ldflags`** — проверка применяемых флагов при сборке пакетов в RPM-окружении.

📋 Интеграция с аудитом и сертификацией

- Для ОС, подлежащих сертификации ФСТЭК/Минобороны, важно зафиксировать применение флагов в **отчётах по защищённости** и при прохождении испытаний.
- В НАЙС.ОС отчёты `checksec` включаются в состав артефактов сборки.
- Возможно применение внешнего аудита (3rd party), особенно для компонентов безопасности, библиотек и служб.

Итог: систематическая проверка — важная часть цепочки доверия. Автоматизация аудита в CI помогает не допустить случайных ошибок и упростить сертификацию решений.

9. Дорожная карта

Защита по умолчанию — это не статичная цель, а постоянный процесс. В НАЙС.ОС мы продолжаем усиливать базовую безопасность системы, добавляя новые технологии и флаги по мере зрелости инструментов и поддержки со стороны toolchain.

Поддержка CET (Control-Flow Enforcement Technology)

CET — аппаратный механизм от Intel, защищающий от атак типа *Return-Oriented Programming (ROP)* и *Jump-Oriented Programming (JOP)* . В планах — сборка ключевых библиотек и исполняемых файлов с флагом `-fcf-protection` и линковкой `-z cet` , с проверкой наличия CET в процессе загрузки.

🛡️ Расширение компиляторных флагов

- **`-fstack-clash-protection`** — защита от переполнения стека на границе страниц (stack clash);
- **`-fcf-protection`** + `-z,cet` — активация CET для компилятора и линковщика;
- **`-D_GLIBCXX_ASSERTIONS`** — дополнительные проверки STL-контейнеров в libstdc++;
- Введение **clang-hardening-профилей** для компонентов, где GCC уступает по статическому анализу и оптимизации.

⚙️ Интеграция с LTO и ASan

В перспективе планируется автоматическое профилирование и сборка **критически важных библиотек** с:

- **LTO (Link-Time Optimization)** для усиления защиты от внутримодульных переходов;
- **ASan (AddressSanitizer)** в режиме тестирования для выявления утечек, OOB-доступов и UAF;
- Поддержка режимов профилирования без лишнего overhead в production-билдах.

Все изменения проходят через систему CI, проверяются на совместимость и тестируются в реальных сценариях, включая сборку пользовательских пакетов и совместимость с сертификационными профилями.

Цель: сделать НАЙС.ОС одной из самых защищённых Linux-систем, где безопасность — не надстройка, а базовое свойство архитектуры.

10. Заключение

В **НАЙС.ОС** безопасность начинается не с антивирусов или файрволов, а с базовой

сборки бинарных файлов. Использование *hardening*-флагов — таких как `PIE`, `RELRO`, `ASLR`, `SSP`, `FORTIFY` — даёт реальную защиту от целого класса уязвимостей на уровне памяти, управления потоком исполнения и целостности данных.

Эти механизмы **включены по умолчанию** для всех системных компонентов НАЙС.ОС, а также рекомендованы для пользовательских и сторонних приложений.

Hardening стоит дёшево, но защищает дорого . Не требует переписывания кода, но даёт выигрыш в стойкости перед потенциальными эксплойтами. Это особенно важно в условиях роста числа supply-chain атак и попыток эксплуатации типовых библиотек.

Мы призываем разработчиков, DevOps-инженеров и системных интеграторов:

- включайте защитные флаги в своих сборках,
- автоматизируйте проверку безопасности на уровне CI,
- расширяйте свою цепочку доверия с самых низких уровней.

Безопасность — это не опция. Это фундамент.