ChatGPT

Разработка программного модуля защиты целостности скрипта ключом — символьным паролем

Введение

В условиях современной цифровой трансформации вопрос обеспечения информационной безопасности становится особенно актуальным. Одной из ключевых задач является защита **целостности программного кода**, особенно скриптов, которые подвержены изменениям как со стороны злоумышленников, так и по ошибке пользователя. Целостность (integrity) наряду с конфиденциальностью и доступностью входит в число фундаментальных свойств информационной безопасности. Она означает сохранность и неизменность информации в процессе ее хранения, обработки и передачи, допускающую изменения данных только с соответствующим разрешением [1]. Нарушение целостности даже в незначительной степени может привести к сбоям в работе программ, утечке конфиденциальных данных или появлению уязвимостей для атак. Поэтому обеспечение целостности кода является критически важным для доверия к программному обеспечению и стабильности системы.

В контексте скриптовых языков (Python, JavaScript, Bash и др.) проблема целостности стоит особенно остро. Скрипты широко используются в автоматизации, веб-разработке, системном администрировании и многих других областях благодаря своей гибкости и простоте. Однако именно эти качества делают их уязвимыми: злоумышленники могут относительно легко прочитать или изменить скрипт, а случайные ошибки – повредить его. Необходимо гарантировать, что исполняемый скрипт не был тайно модифицирован.

Разработка программного модуля защиты целостности скрипта с использованием символьного пароля (ключа) – тема данного реферата – направлена на решение этой задачи. Идея заключается в интеграции криптографической защиты в скрипт при помощи секрета, заданного пользователем в форме пароля. Такой модуль вычисляет криптографический контрольный код (например, HMAC) для скрипта на основе символьного пароля и позволяет впоследствии проверить, не был ли скрипт изменен. Подобный подход сочетает понятность паролей для человека и криптографическую стойкость хеш-функций, обеспечивая **контроль целостности** перед исполнением кода.

В реферате представлены расширенные теоретические сведения о целостности данных и методах ее обеспечения, описаны криптографические основы (хеш-функции, HMAC, цифровая подпись), проведен анализ существующих решений и инструментов для контроля целостности. Основная часть работы посвящена проектированию и реализации модуля на языке Python, использующего HMAC-SHA256 с символьным паролем для защиты скрипта. Приводятся архитектура решения, фрагменты исходного кода, примеры использования и результаты тестирования модуля. Дополнительно рассмотрены вопросы интеграции разработанного подхода в процессы CI/CD и концепцию DevSecOps, анализируются современные атаки на цепочки поставок программного обеспечения и меры противодействия, а также обсуждаются этические и юридические аспекты защиты кода и возможности автоматизации мониторинга целостности.

По итогу демонстрируется, что предложенный модуль может служить эффективным средством предотвращения несанкционированных изменений в скриптах, повышая общую безопасность программных систем. Далее в работе подробно излагаются теоретические и практические результаты, полученные в ходе разработки данного решения.

Глава 1. Теоретические основы защиты целостности

1.1 Понятие целостности данных

Целостность данных — это одно из фундаментальных свойств информационной безопасности, означающее сохранность и неизменность информации в процессе её хранения, обработки и передачи. Суть этого свойства заключается в том, что данные не могут быть случайно или намеренно изменены, удалены или подменены без соответствующего разрешения. Проще говоря, **сохранение целостности** гарантирует, что информация остается точно такой, какой её оставил автор или ответственный владелец, и любое отклонение от исходного состояния может быть обнаружено.

Поддержание целостности информации критически важно для достоверности функционирования программных систем. Любое, даже незначительное изменение данных или кода может привести к некорректной работе программ, утечке конфиденциальной информации или появлению уязвимостей, которыми могут воспользоваться злоумышленники. Например, если в исходный код программы внести незаметную модификацию, меняющую логику проверки прав доступа, это может остаться не замеченным разработчиками, но открыть дверь для несанкционированного доступа.

Целостность данных должна обеспечиваться как при **хранении**, так и при **передаче** информации. В обоих случаях используются различные методы контроля: контрольные суммы, криптографические хеш-функции, коды аутентичности сообщений и цифровые подписи. Так, при хранении файлов целостность можно проверить вычислением и сравнением хеш-суммы файла до и после хранения. При передаче данных по сети часто применяются механизмы, гарантирующие, что принятое сообщение идентично отправленному.

На практике поддержка целостности реализуется через совокупность мер, включая:

- **Вычисление и проверку контрольных сумм** – простейший способ убедиться, что файл не изменился при копировании или хранении.
- **Использование криптографических хеш-функций** (например, SHA-256) – получение "отпечатка" данных, который практически невозможно подобрать без знания исходных данных.
- **Применение HMAC** (код аутентификации сообщения на основе хеша) с секретными ключами – для одновременной проверки целостности и аутентичности данных.
- **Контроль версий и журналирование изменений** – отслеживание всех правок в коде с возможностью отката и выявления несанкционированных изменений.

Если целостность нарушена, это служит серьёзным сигналом. Нарушение может свидетельствовать об ошибке системы, вирусной активности, попытке взлома или случайной порче данных. Именно поэтому контроль целостности часто используется как индикатор возможной атаки или сбоя: при обнаружении несоответствия ожидаемой контрольной суммы система безопасности может сгенерировать тревогу и предотвратить дальнейшее выполнение подозрительного кода.

1.2 Угрозы целостности скриптов

Скрипты – это легковесные исполняемые файлы (на языках Python, JavaScript, Bash и др.), которые широко применяются для автоматизации задач, веб-разработки, системного администрирования и многого другого. Их плюсы – читаемость и простота изменения – одновременно являются источником риска с точки зрения целостности. Рассмотрим основные угрозы, связанные с несанкционированным изменением скриптов:

1. **Внедрение вредоносного кода.** Злоумышленник, получивший доступ к скрипту, может внедрить в него вредоносный фрагмент – например, функцию, отправляющую конфиденциальные данные на внешний сервер, или встроить «логический бомбу»/ шифровальщик. Подобные атаки реализуются через уязвимости в правах доступа (недостаточная защита файловой системы), компрометацию внешних зависимостей или подмену файлов в репозиториях. Результатом может стать скрытая вредоносная функциональность внутри легитимного на первый взгляд скрипта.

2. **Случайное повреждение.** Изменения в скрипте могут произойти непреднамеренно: при ошибочной правке другим разработчиком, из-за сбоя системы или повреждения носителя данных. Даже малейшие неумышленные изменения способны вызвать сбои в логике работы скрипта, утрату части функциональности или, что хуже, открыть лазейки для атак (например, отключив какую-то проверку). Случайный характер таких повреждений усложняет их обнаружение без специальных средств контроля.

3. **Атаки типа "Man-in-the-Middle" (MITM).** Если скрипты передаются по незащищённым каналам (например, скачиваются через HTTP или передаются в открытом виде при клонировании из ненадёжного репозитория), существует риск их подмены в пути. Атакующий, встав на пути передачи, может подменить оригинальный скрипт изменённой версией, включив туда вредоносный код. Получатель, не подозревая, запустит уже модифицированный скрипт. Без проверки целостности обнаружить такую подмену бывает сложно.

4. **Инсайдерские угрозы.** Не только внешние злоумышленники представляют опасность — сотрудник организации с доступом к исходному коду тоже может преднамеренно изменить скрипт, добавив несанкционированную функциональность (например, бэкдор). В корпоративной среде инсайдерские атаки особо опасны, так как исходят от доверенных лиц. Без строгого контроля целостности изменений в репозитории такие правки могут долго остаться незамеченными.

5. **Атаки через зависимости.** Современные скрипты редко работают в вакууме – они используют множество внешних библиотек и пакетов. Если одна из зависимостей скомпрометирована (например, в публичный пакетный репозиторий загружена библиотека с тем же именем, но содержащая вредоносный код, или обновление легитимного пакета было взломано), то целостность конечного скрипта нарушается **без прямого изменения его кода**. Подобные атаки на цепочку поставок ПО стали реальной угрозой: разработчик подключает зависимость, доверяя ей, а та оказывается изменённой злоумышленником.

Нарушение целостности скриптов несет серьезные **последствия**:

- Потеря доверия к программному обеспечению со стороны пользователей или клиентов.

- Утечка конфиденциальной информации (если вредоносный код высылает данные наружу).
- Появление бэкдоров и скрытых уязвимостей, позволяющих повторно компрометировать систему.
- Нарушение работоспособности систем из-за сбоев, вызванных измененным скриптом.
- Финансовые потери и ущерб репутации компании вследствие инцидента безопасности.

Осознавая эти угрозы, организации внедряют меры для защиты целостности скриптов на всех этапах их жизненного цикла – от разработки и тестирования до развертывания и эксплуатации.

1.3 Методы проверки целостности

Для обеспечения и контроля целостности данных и скриптов разработано множество методов, основанных как на некриптографических, так и на криптографических подходах. Цель всех этих методов – своевременно обнаружить любые несанкционированные изменения содержимого скрипта или данных. Ниже перечислены основные способы проверки целостности:

1. **Контрольные суммы (Checksum).** Контрольная сумма – один из простейших способов проверки целостности. С помощью определенного алгоритма (например, CRC32) для файла вычисляется числовое значение, зависящее от содержимого файла. При последующей передаче или хранении файла его контрольная сумма пересчитывается и сравнивается с изначально сохраненной. Несовпадение сумм сигнализирует об изменении. Преимущество контрольных сумм – скорость вычисления и простота реализации. Однако криптографическая стойкость низкая: злоумышленник, изменивший файл, может пересчитать контрольную сумму и подменить ее, сделав изменение незаметным. Таким образом, checksum подходит для обнаружения случайных ошибок передачи, но слабо противостойт целенаправленным атакам.

2. **Криптографические хеш-функции.** Криптографические хеш-функции (SHA-256, SHA-3 и др.) создают уникальный "отпечаток" файла – фиксированной длины строку (хеш), которая существенно меняется даже при изменении одного байта исходных данных. В отличие от простых сумм, криптографические хеши обладают свойством коллизионной устойчивости (крайне сложно подобрать другой вход, дающий тот же хеш) и односторонности (невозможно восстановить исходные данные по хешу). Хеш файла обычно публикуется или хранится в защищенном месте; для проверки целостности вычисляют текущий хеш и сравнивают с эталонным. Этот метод широко используется из-за высокой надежности: подделать хеш без знания исходного файла практически невозможно. Тем не менее сам по себе хеш не защищен от подмены – если атакующий может изменить и файл, и сохранённый хеш, то изменение не будет обнаружено.

3. **HMAC (Hash-based Message Authentication Code).** HMAC представляет собой комбинацию хеш-функции с секретным ключом. Проще говоря, это **код аутентификации сообщения**, вычисляемый через криптографический хеш и секретный ключ. Полученный HMAC прикрепляется к сообщению или файлу. Для проверки получатель также вычисляет HMAC от полученных данных (используя тот же секретный ключ) и сравнивает со снятым значением. Если не совпадает – данные были изменены или ключ неверен. Даже если злоумышленник знает алгоритм хеширования, без владения секретным ключом он не сможет сгенерировать корректный HMAC к поддельному содержимому. Благодаря этому HMAC широко используется для защиты от незаметной подмены данных при передаче по сети и хранении: он гарантирует как целостность, так и подлинность (авторство) данных, поскольку рассчитан с секретом, известным лишь доверенным сторонам.

4. **Цифровые подписи.** Цифровая подпись использует криптографическую пару ключей – приватный (секретный) и публичный. Обладатель приватного ключа вычисляет подпись для файла (например, с помощью алгоритма RSA или ECDSA); любой желающий может проверить эту подпись, зная публичный ключ автора. Цифровая подпись обеспечивает не только целостность (любой изменения данных делает подпись недействительной), но и аутентификацию источника – гарантирует, что файл подписан именно владельцем приватного ключа. Этот метод применяется в системах обновления ПО, в распространении скриптов и программ с повышенными требованиями безопасности (например, подписанные PowerShell-скрипты в Windows). Недостаток – сложность управления ключами (необходимо защищать приватный ключ, распространять публичный) и вычислительные затраты, но для важных случаев безопасности это оправдано.

5. **Контроль версий и мониторинг изменений.** Использование систем контроля версий (Git, Mercurial и др.) позволяет отслеживать все изменения в исходных текстах скриптов. Каждый коммит в Git, например, имеет уникальный хеш, зависящий от содержимого и истории, что обеспечивает целостность версии кода в репозитории. Помимо самого факта изменения, можно видеть, кто и когда внёс правку. В сочетании с процессами code review и автоматизированным аудитом (например, проверками в CI/CD конвейере) контроль версий значительно повышает уверенность в целостности кода на этапе разработки. Однако система версионности не защищает от изменений **вне** репозитория (например, после развертывания на сервере) – там нужны другие механизмы.

6. **Файловый мониторинг (Host-based Intrusion Detection).** Существуют специализированные инструменты для мониторинга целостности файлов в режиме реального времени, такие как *Tripwire*, *AIDE* и др. Они работают следующим образом: при установке системы безопасности создается база (слепок) хеш-сумм всех критичных файлов. Затем агент мониторинга периодически или постоянно проверяет текущие хеши файлов и сравнивает с базовыми. Если обнаруживается расхождение, генерируется оповещение. Такие инструменты позволяют оперативно выявлять пост-фактум изменение исполненных скриптов, конфигураций и других важных файлов. Интеграция с SIEM-системами и центральными консолями безопасности обеспечивает автоматизированное реагирование. Файловый мониторинг – ключевой элемент обнаружения атак на этапе эксплуатации системы.

Перечисленные методы могут комбинироваться для повышения надежности. Например, разработчик может использовать контроль версий во время разработки, а при деплое в продакшен – генерировать HMAC для скрипта и затем на сервере иметь настроенный файловый мониторинг, который проконтролирует целостность развернутых скриптов. В следующих разделах рассматривается выбранный в рамках данной работы метод защиты и его криптографические основы.

1.4 Роль криптографии в обеспечении целостности

Криптография играет решающую роль в современных методах защиты целостности информации. В отличие от простых контрольных сумм, криптографические примитивы обеспечивают стойкость против намеренных подделок и глубокий уровень доверия к данным. Основные криптографические инструменты, задействованные для контроля целостности, включают хеш-функции, коды аутентичности сообщений (HMAC) и цифровые подписи.

Главное свойство криптографических методов – их устойчивость к нападениям злоумышленников. Простейший подход к проверке целостности (например, хранение копии

файла и побайтовое сравнение) становится непрактичным при больших объемах данных и не защищает от умышленной подмены. Криптографические же **хеш-функции** позволяют вычислить компактный идентификатор содержимого, который практически невозможно изменить преднамеренно так, чтобы получился тот же самый хеш (коллизия крайне маловероятна). Это значит, что любое изменение данных будет с высокой вероятностью обнаружено при сравнении хешей.

Тем не менее, простого хеша недостаточно, если атакующий способен его пересчитать. Вот тут вступают в силу **ключевые методы**. HMAC, как было описано выше, привносит секретный ключ в процесс хеширования. Без знания ключа злоумышленник, изменивший файл, не сумеет получить корректный HMAC, и проверка провалится. Таким образом, криптография добавляет элемент секрета, отсутствующий в неключевых методах, и обеспечивает контроль как целостности, так и подлинности (авторства) данных – ведь только владелец ключа мог сгенерировать правильный код аутентичности.

Цифровая подпись решает сходную задачу, но с помощью асимметричной криптографии. Она не требует предварительного обмена секретом между сторонами проверки: достаточно, чтобы все доверяющие целостности имели доступ к подлинному публичному ключу подписанта. Криптографические подписи повсеместно используются для гарантии целостности программного обеспечения – например, дистрибутивы Linux подписываются ключами разработчиков, сертификаты приложений позволяют ОС проверять, что исполняемый файл не изменен с момента подписания. Эти меры опираются на стойкость алгоритмов, официально утвержденных стандартами (RSA, ECDSA, ГОСТ Р 34.10-2012 и т.д.), и юридически значимы, так как часто приравниваются к ручной подписи документа.

Наконец, криптография обеспечивает и **защиту хранимых контрольных кодов**. Например, файл с хеш-суммами сам может быть подписан или защищен HMAC’ом, база данных контроля целостности в инструментах типа Tripwire шифруется и подписывается, чтобы злоумышленник не смог её тихо подделать. Таким образом, создается многоуровневая защита: даже если один уровень будет скомпрометирован, другой сработает как сетка безопасности.

Подводя итог, криптография является фундаментом прочных систем контроля целостности. Без нее современные методы вроде HMAC и цифровых подписей были бы невозможны, а злоумышленники легко обходили бы простейшие проверки. Криптографические алгоритмы при правильном использовании гарантируют, что обнаружить несанкционированное изменение – лишь вопрос времени, и значительно повышают уверенность в сохранности и корректности кода и данных.

Глава 2. Криптографические основы

2.1 Хеш-функции и их свойства

Криптографическая хеш-функция – это алгоритм, который по произвольному входному сообщению вычисляет битовую строку фиксированной длины, называемую *хешем* (или дайджестом). Ключевые свойства хорошей хеш-функции:

- **Односторонность:** имея значение хеша, невозможно практически восстановить оригинальные входные данные (криптоаналитически это должно требовать перебора всех возможных входов).

- **Стойкость к коллизиям:** крайне трудно найти два разных сообщения, которые дают одинаковый хеш. Более строго, невозможно подобрать к заданному сообщению другое так, чтобы их хеши совпали (стойкость к коллизиям второго рода), и невозможно сконструировать пару разных сообщений с одинаковым хешем (первого рода) за приемлемое время.
- **Свойство лавины:** небольшое изменение входного сообщения (например, изменение 1 бита) приводит к непредсказуемому кардинальному изменению выходного хеша (статистически изменяется примерно половина битов хеша). Это усложняет задачу злоумышленнику, пытающемуся понять, как повлиять на хеш целенаправленно.

Классические примеры криптографических хеш-функций – семейства SHA-2 (SHA-256, SHA-512 и др.), SHA-3, ранее MD5, SHA-1 (последние два устарели и не рекомендуются из-за найденных коллизий). Стандарт **SHA-256** (Secure Hash Algorithm с размером выхода 256 бит) сегодня один из наиболее популярных для контроля целостности данных [2] [3]. Его использование оправдано балансом между скоростью и стойкостью: 256-битный хеш практически нереально подобрать перебором (число возможных комбинаций 2^{256} астрономически велико).

Важно понимать, что хеш-функция не требует секретного ключа – она является *неключевым* алгоритмом. Это означает, что любой может вычислить хеш от заданных данных. Применительно к контролю целостности это палка о двух концах: с одной стороны, любой проверяющий может самостоятельно убедиться, что, например, скачанный файл совпадает по хешу с опубликованным на сайте разработчика (частый сценарий для дистрибутивов ПО). С другой стороны, злоумышленник, подменив файл, может так же вычислить новый хеш и подделать опубликованное значение. Поэтому хеши сами по себе часто публикуют по защищенным каналам или дополняют электронной подписью.

Пример использования хеша: пускай имеется скрипт `script.py`. Вычисляем его SHA-256 хеш, получаем, например, `d2d5...9af3` (64 шестнадцатеричных символа). Сохраняем это значение. При подозрении на изменение скрипта вычисляем снова SHA-256 – если значение отличается от `d2d5...9af3`, значит файл был изменен. В противном случае с высокой вероятностью файл сохранил целостность. Вычисление хешей эффективно даже для больших файлов: современные процессоры могут хешировать сотни мегабайт в секунду, поэтому для типичных скриптов (сотни КБ) это доли секунды работы.

2.2 Символьные пароли и генерация ключей

В криптографических системах под *ключом* понимается некоторая секретная информация (набор бит определенной длины), используемая в алгоритмах шифрования или вычисления кодов аутентичности. Однако человек оперирует символами, поэтому на практике ключ часто задается в виде **символьного пароля** – строки, придуманной и запомненной пользователем. Примеры: `"correctBatteryHorseStaple!"` или `"P@ssw0rd123"` и т.п. Пароль легче ввода и запоминания, чем случайная последовательность бит.

Возникает задача преобразования такого пароля в криптографический ключ. Пароль может иметь переменную длину и недостаточную энтропию (не случайный, а осмысленный, потому потенциально уязвимый для перебора), в то время как алгоритму обычно требуется ключ фиксированной длины, например 256 бит, с максимально равномерным распределением.

Генерация ключа из пароля обычно включает процедуры *хеширования или деривации ключа*. Самый простой подход – взять криптографический хеш от строки пароля. Например, SHA-256 от `"password"` даст 256-битное значение, которое можно использовать как ключ. Однако простой

хеш не учитывает возможности подбора паролей по словарю. Поэтому часто применяются специализированные алгоритмы **PBKDF (Password-Based Key Derivation Function)** – функции вытяжки ключа из пароля, которые включают в себя:

- **Соль (salt):** случайная строка, добавляемая к паролю перед деривацией, чтобы затруднить атаки с использованием заранее вычисленных таблиц (rainbow-tables). Соль делает каждый пароль уникальным в контексте деривации.
- **Итерацию алгоритма многократно:** чтобы увеличить вычислительную сложность подбора. Алгоритм может хешировать пароль с солью тысячи и миллионы раз подряд (параметр итераций), замедляя проверку одного пароля. Для легитимного пользователя небольшая задержка при ввода пароля не критична, а перебор большого числа возможных паролей резко затрудняется.
- **Конечное преобразование в нужный размер ключа:** например, PBKDF2-HMAC-SHA256 может выдать ключ длиной 256 бит.

В рамках нашей задачи по защите скрипта может быть применен и упрощенный подход, так как предполагается, что пароль задается ответственным пользователем и храниться нигде не будет – он просто вводится для генерации и проверки. Мы можем использовать сам пароль (его двоичное представление) как секретный ключ для HMAC. Библиотека `hmac` в Python, например, позволяет передать любой байтовый секрет. Тем не менее, лучше, если пароль будет удовлетворять требованиям стойкости (достаточная длина, наличие различных типов символов) – это противодействует попыткам подбора ключа методом перебора (bruteforce).

Еще один момент – **хранение ключа**. В проектируемом модуле ключ никуда не сохраняется (во избежание его компрометации), а вводится пользователем вручную при генерации подписи и при проверке. Таким образом, безопасное управление ключом возлагается на человека (он должен не разглашать пароль). Это несколько упрощает реализацию системы, но переносит ответственность за секрет на пользователя. В более сложных системах используются менеджеры секретов, аппаратные хранилища ключей (токены) и т.д., но об этом мы поговорим в перспективных направлениях развития.

Вывод: символьный пароль может служить практичным средством задания секретного ключа для HMAC, однако необходимо преобразовать его подходящим образом (например, хешированием). В нашем модуле мы будем использовать алгоритм HMAC-SHA256 напрямую с введенным паролем, полагаясь на то, что пользователь выберет достаточно надежный пароль. При необходимости решение можно модернизировать, добавив PBKDF2 для усиления ключа из пароля.

2.3 Алгоритмы HMAC и цифровой подписи

HMAC (Keyed-Hash Message Authentication Code) – алгоритм вычисления кодов аутентичности на основе хеша – играет центральную роль в нашем модуле. Как отмечалось ранее, HMAC сочетает в себе хеш-функцию и секретный ключ. Стандартный алгоритм HMAC определяется в RFC 2104 [2] и в FIPS 198. Он принимает на вход сообщение и ключ и выдаёт MAC (Message Authentication Code) фиксированной длины. Принцип работы HMAC можно описать упрощенно так:

1. Ключ, если он длиннее блока (например, >64 байт для SHA-256), хешируется и/или дополняется нулями до размера блока.
2. Формируются две производные от ключа: *inner key* и *outer key* путём наложения двух разных битовых масок (определённых константами opad и ipad).

3. Вычисляется внутренний хеш: `Hash(inner_key || message)`, где `||` означает конкатенацию.
4. Вычисляется финальный HMAC как `Hash(outer_key || inner_hash)`.

Такая структура обеспечивает, что знание хеша сообщения не позволяет вычислить HMAC без ключа. Корректность HMAC легко проверяется тем же образом: принимающая сторона (или проверяющий модуль) берёт свое копие ключа, повторно вычисляет HMAC сообщения и сравнивает с переданным значением. Совпадение означает, что сообщение не изменилось и код был сгенерирован обладателем ключа (т.е. аутентичен).

Преимущество HMAC в контексте целостности скриптов – простота и скорость. Например, HMAC-SHA256 вычисляется с примерно той же скоростью, что и сам SHA-256 (накладные расходы минимальны), и уже давно есть готовые реализации во многих языках (в Python – модуль `hmac`). При этом безопасность HMAC опирается на прочность используемой хеш-функции и секретность ключа. При условии достаточной длины пароля (ключа) и стойкости хеша (SHA-256) считается, что HMAC защищен даже от криптоанализа, и более того, HMAC остается криптостойким даже если основная хеш-функция имеет некоторые слабости в коллизиях [3] (например, HMAC-MD5 все еще не тривиален для взлома, несмотря на уязвимости MD5, за счет добавления ключа).

Цифровая подпись – другой столп криптографии, часто используемый для контроля целостности, особенно когда нужно подтвердить авторство. В отличие от HMAC, где используется один ключ на две стороны, системы цифровой подписи используют пару ключей: открытый (public) и закрытый (private). Алгоритмы цифровой подписи (RSA, DSA, ECDSA, Ed25519 и др.) работают так:

- Генерируется ключевая пара. Приватный ключ сохраняется в тайне у владельца (подписанта), публичный ключ распространяется для всех заинтересованных.
- Для файла (сообщения) вычисляется хеш-значение (чаще всего SHA-256 или SHA-3), затем этот хеш шифруется или используется в формуле с приватным ключом – получается подпись (набор бит, обычно сопоставимый по размеру с ключом, например 256 бит или больше).
- Подпись публикуется или прикрепляется к файлу. Любой проверяющий может взять соответствующий публичный ключ, тот же хеш-функцию и проверить подпись – т.е. удостовериться, что она действительно была создана владельцем приватного ключа для данного сообщения.

Для **контроля целостности** цифровая подпись полезна тем, что если сообщение изменено, даже если злоумышленник обладает публичным ключом, он не сможет сгенерировать валидную подпись (не зная приватного). Проверка подписи при несоответствии хеша укажет на нарушение целостности. Этот подход широко применяется: например, пакеты обновлений ОС подписываются разработчиком, и система перед установкой проверяет подпись, тем самым предотвращая установку поддельных обновлений. В веб-разработке JSON Web Tokens (JWT) могут подписываться для защиты целостности токена.

Однако для нашей задачи – защиты самостоятельного скрипта – внедрение полной инфраструктуры цифровой подписи может быть избыточным. Нужно было бы генерировать пару ключей, хранить приватный ключ где-то безопасно, а публичный включать в программу или распространять. Это сложнее, чем использование общего секрета (пароля) в ограниченном окружении. Поэтому выбор в проекте сделан в пользу HMAC с симметричным ключом (паролем).

Тем не менее, понимание алгоритмов цифровой подписи важно для общей картины: в случаях, когда скрипт распространяется широкому кругу пользователей, безопаснее применить именно подпись. Тогда каждый пользователь сможет проверить целостность скрипта по публичному ключу автора, не обмениваясь с ним секретами. Это часто применяется в open-source: автор выкладывает скрипт и файл `.asc` (PGP-подпись), а скачавшие проверяют подпись с помощью публичного PGP-ключа автора.

В заключение, HMAC и цифровые подписи – взаимодополняющие инструменты. **HMAC** проще и быстрее, подходит для доверенных сред (где можно поделиться секретом), обеспечивает целостность и скрытность (значение HMAC не раскрывает ничего о данных без ключа). **Подпись** же незаменима в открытых экосистемах для подтверждения авторства. В нашей реализации используется HMAC, но архитектура модуля может быть адаптирована и под цифровые подписи при необходимости.

Глава 3. Анализ существующих решений

3.1 Обзор библиотек и инструментов

Перед разработкой собственного модуля имеет смысл рассмотреть существующие решения для защиты или проверки целостности скриптов и файлов, чтобы учесть их возможности и ограничения. На рынке и в сообществе существуют как специализированные инструменты, так и библиотеки общего назначения, связанные с нашей задачей:

- **Встроенные средства языков и ОС.** Некоторые операционные системы и интерпретаторы предусматривают механизмы контроля целостности. Например, Windows PowerShell позволяет *подписывать* скрипты с помощью сертификатов; при включенной политике Execution Policy система откажется запускать неподписанный или изменённый после подписания скрипт. В Unix-средах нет строгого требования подписи скриптов, но существуют утилиты вроде `sha256sum` для вычисления хешей файлов вручную. В языке Python в стандартной библиотеке есть модули `hashlib` и `hmac` для вычисления контрольных хешей и HMAC, однако готового высокого уровня решения «self-integrity check» для скриптов нет – эту задачу обычно решают на уровне упаковки приложений.

- **Сторонние библиотеки для Python.** В экосистеме Python можно найти библиотеки, упрощающие расчёт и проверку хешей. Например, `pyHMAC` (условно, если бы существовал) или просто использование `hmac.new()` с нужными параметрами. Но важнее другое: существуют менеджеры пакетов и инструменты развёртывания, которые обеспечивают целостность зависимостей. *pip* при установке пакета из индекса PyPI может проверять хеш скачанного дистрибутива (`pip install --require-hashes ...`), сравнивая с заранее известными хешами в файле `requirements.txt`. Это предотвращает подмену пакетов. Однако сами скрипты проекта pip не проверяет – предполагается, что разработчик доверяет своему исходному коду.

- **Системы контроля версий и CI-инструменты.** Git не зря упоминался – он гарантирует неизменность истории коммитов (любая модификация в прошлом изменит хешы всех следующих коммитов). Это не защита от изменения *развернутого* скрипта, но защита на этапе разработки. Некоторые CI/CD инструменты, такие как Jenkins, GitLab CI, могут сохранять артефакты сборки вместе с хеш-суммами или подписями. Например, можно настроить pipeline, который после сборки (или тестирования) публикует SHA-256 всех выходных файлов. Это облегчает последующую проверку на серверах.

- **Инструменты мониторинга целостности (HIDS).** Выше были упомянуты Tripwire, AIDE – это класс программ *Host-based Intrusion Detection Systems* (HIDS) с функцией File Integrity Monitoring (FIM). Они в основном ориентированы на администраторов систем: позволяют контролировать, что на сервере или рабочей станции никто посторонний не менял критические файлы. Tripwire изначально был open-source *UNIX*-утилитой, ныне превратился в коммерческий продукт, но есть его открытые аналоги (AIDE – Advanced Intrusion Detection Environment). Для нашего узкого случая их применение возможно: администратор может включить папку со скриптами в контроль, и тогда система будет предупреждать о любом изменении. Недостатком является общая направленность: HIDS следит за всем подряд, а не конкретно интегрирован со скриптом. Наш модуль, встроенный в сам скрипт, мог бы, напротив, выполнять самопроверку непосредственно при запуске.

- **Средства развертывания и упаковки.** В сфере доставки кода существуют решения, косвенно влияющие на целостность. Например, Docker-контейнеры: образ контейнера может быть подписан (Docker Content Trust с использованием подписей на базе ключей RSA/ECC). Если скрипт распространяется как часть контейнера, то подпись контейнера обеспечивает и его целостность. Также, в более широком смысле, системы управления конфигурациями (Ansible, Chef, Puppet) могут иметь механизмы валидации хешей для файлов, которые они раскатывают на сервера (например, в Ansible есть параметр `checksum` для модуля `copy`, гарантирующий, что файл скопирован без искажений).

- **Специализированные утилиты для разработчиков.** Существуют скрипты и утилиты, которые разработчики пишут самостоятельно для проверки целостности. Например, некоторыми сообществами практикуется публикация *checksum-файлов* наряду с дистрибутивами (файл `.sha256` с перечнем имен файлов и их хешей). Затем простая утилита или команда проверяет, что скачанные файлы соответствуют заявленным хешам. Это не требует сложной криптографии, но повышает уверенность в отсутствии случайной порчи. В случае злонамеренной подмены, правда, такие файлы должны быть тоже защищены (например, подписаны).

Подводя итог обзора: для прямой задачи "обнаружить изменение скрипта" широко распространенных узкоспециализированных библиотек нет – обычно разработчики применяют общие криптографические инструменты (хеши, HMAC, подписи) либо полагаются на инфраструктурные решения (контроль версий, CI, FIM-системы). Это подтверждает актуальность разработки нашего модуля: он заполняет нишу между тяжелыми системами мониторинга и ручными проверками, предлагая встроенное в скрипт решение.

3.2 Сравнение подходов по критериям безопасности, производительности, удобству

Рассмотрим основные подходы контроля целостности применительно к скриптам и сравним их по нескольким критериям:

Подход	Безопасность (стойкость к атакам)	Производительность	Удобство использования
Простая контрольная сумма (CRC и пр.)	Низкая: легко подделать, зная алгоритм. Не используют секретов или ключей, уязвимо к преднамеренным изменениям.	Очень высокая скорость (сотни МБ/с); накладные расходы пренебрежимы.	Простая реализация, есть утилиты по умолчанию. Но не годится для защиты от злоумышленника, только для случайных ошибок.
Крипто-хеш (SHA-256 и пр.)	Высокая: сложно найти изменение, дающее тот же хеш. Но если злоумышленник может пересчитать хеш и заменить его хранилище, атака возможна.	Высокая: вычисление SHA-256 для небольших файлов занимает миллисекунды.	Широко поддерживается (библиотеки, утилиты). Требует хранения эталонного хеша в безопасном месте.
HMAC (с секретным ключом)	Очень высокая: без знания ключа подменить данные так, чтобы сошёлся HMAC, практически невозможно. Защищает от атак типа "подмена с пересчетом".	Высокая: скорость сопоставима с хеш-функцией (небольшая дополнительная обработка ключа).	Необходимо безопасно распределить/ хранить секретный ключ между генератором и проверяющим. В нашем случае пароль вводится вручную – удобно, но для массового внедрения был бы вопрос.
Цифровая подпись	Очень высокая: основана на стойкости асимметричных алгоритмов. Требует скомпрометировать приватный ключ для атаки (или найти коллизию хеша, что практически нереально).	Ниже, чем у хешей/ HMAC: генерация и проверка подписи медленнее (особенно RSA при больших ключах). Однако для небольших скриптов разница незначительна.	Необходимо управление ключами (хранение приватного, распространение публичного). Проверка требует дополнительного ПО (например, GPG) или встроенной поддержки.

Подход	Безопасность (стойкость к атакам)	Производительность	Удобство использования
Системы контроля версий	Высокая на этапе разработки: злоумышленнику сложно скрыть несанкционированную правку в репозитории. Но после выведения из репо (на боевой сервер) – не применима.	Повсеместное использование SCM не замедляет работу программиста. Накладные расходы на хеширование при коммите невидимы для пользователя.	Очень удобно для команды разработчиков, автоматически документирует изменения. Не рассчитано на проверку целостности на клиентской стороне или в продакшене без дополнительных средств.
Файловый мониторинг (Tripwire/ AIDE)	Высокая: при правильной настройке и защите базы хешей выявит любые изменения файлов. Однако злоумышленник может попытаться отключить или подменить сам монитор, поэтому FIM должен работать в защищённой среде.	Требует периодических сканирований или постоянного отслеживания изменений в ФС. Может загружать систему при большом числе контролируемых файлов, но для нескольких скриптов нагрузка минимальна.	Настройка требует навыков, интеграция с реагированием – отдельная задача. Обычным пользователям неудобна, ориентирована на администраторов.

Из таблицы видно, что **HMAC** выступает привлекательным компромиссом для нашей задачи: он обеспечивает практически ту же степень защиты, что и цифровая подпись (в условии ограниченного круга доверенных пользователей с секретом), но проще в реализации и быстрее в работе. Простое хеширование недостаточно защищено, а контрольные суммы вовсе небезопасны от умышленных изменений.

С точки зрения удобства, конечному пользователю (разработчику скрипта или администратору) наш модуль должен предоставить интуитивный интерфейс: указать файл скрипта, задать пароль, получить сохранённую подпись; затем в любое время проверить файл тем же паролем. Это гораздо проще, чем разворачивать полный Tripwire или настраивать процесс цифровой подписи, особенно в небольших проектах.

Еще один критерий – **встроенность в процесс разработки/развёртывания**. Здесь HMAC-модуль выигрывает тем, что его можно запускать автоматически, например, интегрировать в *git hook* (при каждом коммите или перед деплоем генерировать/проверять HMAC) или в *CI-пайплайн*. Цифровая подпись тоже допускает автоматизацию, но требует инфраструктуры (сертификаты, либо PGP-ключи и доверие к ним). Файловый мониторинг скорее работает постоянно на серверах, чем является частью CI.

Таким образом, сравнение подходов подтверждает выбор **HMAC-SHA256 с символьным паролем** как основы для разрабатываемого модуля – он обеспечивает высокий уровень безопасности при относительной простоте внедрения и использования.

3.3 Выводы и выбор метода

Анализ показал, что наиболее подходящим методом для контроля целостности скрипта в рамках наших требований является применение **криптографического HMAC** с секретным ключом (паролем). Данный метод удовлетворяет критериям безопасности (стойкость к подделке без знания секрета), имеет небольшое влияние на производительность и может быть реализован с использованием стандартных библиотек (что снижает количество потенциальных новых уязвимостей, т.к. задействуется отлаженный код).

Другие рассмотренные варианты либо не обеспечивают должного уровня защиты (просто хеширование или контрольные суммы без ключа), либо усложняют систему (необходимость управлять ключевой инфраструктурой для цифровой подписи), либо выходят за рамки задачи (мониторинг целостности общего назначения). Конечно, в специфических сценариях, например при массовом распространении скриптов сторонним пользователям, целесообразно в перспективе добавить поддержку цифровых подписей. Однако базовый модуль, реализованный с HMAC, уже может быть встроен в такие сценарии: достаточно генерировать подпись с приватным ключом и проверять публичным – архитектурно изменение будет минимальным.

Итак, выбираемый подход – **HMAC-SHA256 с символьным паролем в качестве ключа**. Далее в работе мы детально разберем проектирование модуля на этой основе и опишем процесс его реализации и тестирования.

Глава 4. Проектирование программного модуля

4.1 Постановка задачи и требования

Постановка задачи: Необходимо разработать автономный модуль, который позволяет генерировать и проверять код, удостоверяющий целостность заданного скрипта, используя секретный ключ в виде символьного пароля. Модуль должен быть реализован программно (на выбранном языке), легко интегрироваться в существующие процессы и обеспечивать высокую надежность обнаружения изменений в файлах скриптов.

Исходя из этого, сформулируем основные **требования к функциональности и безопасности**:

- Модуль должен уметь вычислять защитный код (подпись) для заданного файла скрипта на основе пароля. Этот код должен явно зависеть от содержимого файла и от пароля.
- Модуль должен уметь проверять файл: сравнивать его текущий защитный код с ранее вычисленным (хранимым где-то) и выдавать результат проверки (целостность подтверждена или нарушена).
- В случае несоответствия содержимого или неправильного пароля проверка должна сообщать об этом факте (что важно – не разглашая лишней информации о пароле).
- Решение должно быть кроссплатформенным, не зависеть от специфики ОС, чтобы скрипты могли проверяться в разных средах (например, на Windows и Linux одинаково).
- Желательно минимизировать внешние зависимости и обеспечить простоту установки/использования модуля. В идеале – использовать только стандартные средства языка.

- **Безопасность:** пароль не должен сохраняться на диске в открытом виде; подпись должна храниться таким образом, чтобы изменение файла без ее соответствующего обновления стало очевидным (например, хранить отдельно).
- Следует предусмотреть защиту от типичных ошибок: например, исключить из расчета подписи саму подпись (если она встраивается в файл), чтобы при проверке не возникал парадокс; обрабатывать различные кодировки файлов, чтобы одинаковое содержимое дало один и тот же код.
- Удобство: интерфейс запуска (CLI или функция) должен быть понятным – например, команды `generate` и `verify` с нужными аргументами. Также ошибки (несуществующий файл, неправильный формат подписи и т.д.) должны трактоваться корректно.

Уточнение контекста использования: Предполагается, что модульом будут пользоваться разработчики или администраторы. Возможные сценарии: - Разработчик после написания скрипта генерирует для него подпись и, например, отправляет скрипт вместе с файлом подписи (.sig) получателю. Получатель запускает модуль в режиме проверки, вводит пароль (который ему передан по защищенному каналу), и убеждается, что скрипт не поврежден. - Администратор разворачивает на сервере скрипты и с помощью нашего модуля встроенно проверяет их целостность на старте (самопроверка: скрипт может запускать модуль как подмодуль, проверяя сам себя). - Интеграция с CI: конвейер сборки при деплое вычисляет подписи всех скриптов и сохраняет где-то; перед запуском в продакшене автоматически проверяется целостность.

Исходя из таких сценариев, **требования к надежности и простоте интеграции** высоки. Модуль должен корректно обрабатывать десятки и сотни файлов (например, в проекте с множеством скриптов). Также стоит заложить возможность расширения – например, заменить алгоритм хеширования на более новый при необходимости или добавить опцию использования асимметричной подписи.

4.2 Архитектура модуля

Архитектура программного модуля защиты целостности скрипта ориентирована на простоту интеграции и модульность. Модуль должен одинаково хорошо работать автономно (через командную строку) и вызываться из других систем или скриптов (например, как библиотечная функция). При проектировании учтены принципы расширяемости и явного разделения ответственности между компонентами.

На верхнем уровне можно выделить следующий **принцип работы HMAC-модуля** (workflow):

1. Пользователь (или процесс CI) указывает файл скрипта и задает пароль (секретный ключ).
2. Модуль читает содержимое файла, при необходимости удаляя из него ранее встроенную подпись (чтобы не повлиять на расчет).
3. На основе содержимого и пароля вычисляется HMAC (например, алгоритмом HMAC-SHA256).
4. Полученное значение сохраняется – либо в отдельный файл подписи, либо прямо в конец скрипта (в виде специального комментария).
5. При проверке целостности модуль снова читает файл, извлекает или получает сохраненный эталонный HMAC, вычисляет новый HMAC и сравнивает их.
6. По результатам сравнения выводится результат: **«Целостность подтверждена»** (если совпало) или **«Файл изменён»** (если нет совпадения), а также соответствующий код возврата/исключение для автоматизированных сценариев.

С учетом этого процесса, разобьем модуль на несколько логических **компонентов**:

- **Компонент чтения скрипта:** отвечает за загрузку содержимого указанного файла. Должен корректно обрабатывать чтение в бинарном режиме (на случай, если скрипт содержит байтовые данные), либо в текстовом с правильной кодировкой (UTF-8 по умолчанию). Также именно здесь выполняется фильтрация встроенных хешей, если скрипт содержит в конце свой HMAC в комментарии – чтобы при повторном вычислении не учитывать этот уже сохраненный код. Этот компонент также должен обрабатывать ошибки чтения (например, файл не найден, нет прав доступа) и возвращать содержимое файла в стандартизированной форме (например, bytes).

- **Компонент генерации HMAC:** получает на вход бинарные данные скрипта и символьный пароль от пользователя. С помощью выбранного алгоритма (HMAC-SHA256) вычисляет хеш-значение. Реализуется через библиотеку `hmac` (в Python) или аналогичные средства в другом языке. Этот компонент инкапсулирует логику деривации ключа из пароля (если требуется, например, предварительно хешировать пароль), а также формирует итоговую строку подписи в удобном формате (например, в шестнадцатеричном представлении или Base64 для компактности).

- **Компонент верификации:** на этапе проверки целостности сравнивает текущий вычисленный HMAC с ранее сохранённым. Ранее сохраненное значение этот компонент получает либо из отдельного файла, либо как часть данных (например, считанное из конца проверяемого файла). Он отвечает за выдачу понятного результата проверки. Кроме того, этот компонент может вести логирование или формировать отчет (например, записывать в лог-файл факт успешной или неуспешной проверки с меткой времени, что полезно для аудита безопасности).

- **Компонент сохранения/встраивания подписи:** при генерации отвечает за запись вычисленного HMAC-строки либо в отдельный файл подписи (например, с тем же именем и расширением `.sig`), либо за добавление его в конец исходного скрипта в виде комментария. Должен обеспечить, чтобы запись не нарушала работоспособность скрипта (поэтому формат комментария подбирается в зависимости от типа файла: для .py и .sh – `# ...`, для .js – `// ...`). Также должен избегать дублирования – например, если скрипт уже содержит подпись, её надо обновить, а не дописать еще одну.

- **Компонент командного интерфейса (CLI):** предоставляет пользователю удобный интерфейс для вызова вышеперечисленных функций. Обычно это разбор аргументов командной строки. Предусмотрены команды:

- `generate` — для генерации подписи указанного файла (с параметрами: путь к файлу, пароль, возможно, выбор способа сохранения – файл или комментарий).
- `verify` — для проверки целостности (с параметрами: путь к файлу, пароль, путь к файлу подписи если отдельный).
- В командной строке также поддерживаются флаги/опции: например, `--output` для указания имени файла подписи, `--quiet` для минимального вывода, `--algo` для выбора алгоритма (если захотим поддержать SHA-512, например).

Каждый компонент можно реализовать как функцию или класс. В рамках архитектуры на языке Python логично разделить их по модулям. Например, структура проекта может выглядеть следующим образом:

```
hmac_guard/
├── main.py              # CLI-интерфейс (разбор аргументов, вызов нужных функций)
├── hasher.py            # Логика генерации HMAC и работы с ключами
├── file_handler.py      # Работа с чтением файла, извлечением/встраиванием подписи
├── verifier.py          # Функции сравнения и проверки подписи
└── utils.py             # Вспомогательные функции (например, форматирование строки, логирова
```

(Эта структура приведена для примера и соответствует компонентам; в реализации может быть объединена иначе, но концептуально поможет разделить обязанности.)

Технологии, выбранные для реализации: - **Язык:** Python 3.x – он кроссплатформенный, удобный для работы со скриптами, имеет богатую стандартную библиотеку. - **Библиотеки:** используем стандартные `hashlib` (для хешей SHA-256), `hmac` (для удобного вычисления HMAC) и `argparse` (для разбора аргументов CLI). Дополнительно `os` для работы с файлами, `sys` для вывода и получения информации о потоках, возможно `logging` для логирования.

Таким образом, архитектура модуля строится вокруг четкого потока данных: файл -> чтение -> хеширование с ключом -> вывод/сохранение кода; и аналогично верификация: файл + код -> пересчет -> сравнение -> результат. Такой дизайн обеспечивает простоту понимания и дальнейшей поддержки. В следующем разделе мы перейдем к практической реализации этих компонентов.

4.3 Структура хранения контрольных сумм

Одним из вопросов проектирования было: **где хранить контрольное значение (HMAC) скрипта?** Рассмотрим два подхода, оба из которых мы решили поддержать для гибкости:

- **Отдельный файл подписи.** В этом варианте при генерации для `script.py` создается файл, например `script.py.sig`, содержащий вычисленное значение HMAC (и, возможно, идентификатор алгоритма). Преимущество такого подхода – исходный скрипт остается совершенно неизменным (что важно, если, скажем, он под цифровой подписью или его хеш хранится еще где-то). Подпись можно хранить централизованно, например, все `.sig` файлы в защищенной директории. При проверке модулю нужно знать путь до файла подписи. Недостаток – нужно не потерять файл `.sig` и обеспечить его синхронизацию с самим скриптом (например, при копировании скрипта не забыть скопировать и `.sig`).

- **Встраивание подписи в файл.** Здесь модуль дописывает HMAC внутрь самого скрипта, как комментарий. Например, для Python-скрипта это может выглядеть так:

  ```
  # HMAC: 3f2e5d...abc1 (алгоритм: SHA256)
  ```

  помещенный в конце файла. При таком подходе проверка упрощается – модуль сам находит эту строку и извлекает из нее ожидаемое значение. Пользователю не нужно управлять двумя файлами. Минус – файл скрипта изменяется (нужно учесть, что это не влияет на его выполнение, поэтому важно писать именно как комментарий). Еще один нюанс: при последующих генерациях подписи старую надо обновлять, а не дублировать, поэтому компонент сохранения подписи должен уметь удалять старую строку. Также,

злоумышленник, модифицируя скрипт, может попытаться обновить подпись подделкой – но без знания пароля он не сможет сгенерировать корректное значение, так что такой подлог обнаружится (подпись просто не будет соответствовать контенту).

Мы планируем позволить пользователю выбрать метод: по умолчанию, скажем, **отдельный файл** (более традиционно), а опционально – `--embed` флаг для встроенного комментария. Технически и то, и другое несложно реализовать.

Формат хранения: HMAC длиной 32 байта (для SHA-256) удобно представлять в виде 64 символов шестнадцатеричного числа. Можно дополнительно сократить (Base64 длиной ~44 символа), но hex читаем и достаточно компактен. Также стоит приписывать идентификатор алгоритма и версию формата. Например, строка в `.sig` файле:

```
HMAC-SHA256:3f2e5d...abc1
```

При добавлении новых алгоритмов или версии модуля это поможет понимать, чем была получена подпись.

Защита файла подписи: Сам по себе `.sig` файл или комментарий не секретен (он не раскрывает пароль, т.к. HMAC не обратим), но важно предотвращать его несанкционированное изменение. Если злоумышленник мог изменить и скрипт, он может и подпись (в случае отдельного файла – заменить на свой, в случае комментария – изменить строку). Однако без знания пароля вероятность угадать корректное HMAC пренебрежимо мала (2^{256} вариантов). Тем не менее, если рассуждать о стойкости: атакующий может попытаться **отключить проверку**, например, удалив строку подписи или `.sig` файл. Поэтому следует предусмотреть, что отсутствие подписи при проверке трактуется как нарушение целостности (или хотя бы вызывает предупреждение). Наш модуль при команде `verify` будет сообщать, если ожидаемой подписи не найдено: "Подпись отсутствует – целостность не может быть подтверждена". Это тоже важная информация для безопасности.

Таким образом, структура хранения контрольных сумм является частью общей архитектуры безопасности. Мы делаем выбор в пользу гибкости и простоты: читаемый формат, наличие опций, и строгая реакция на любые несоответствия (не найдена подпись = проблема, подпись не совпала = проблема).

4.4 Выбор алгоритма: HMAC-SHA256 с символьным паролем

Выше мы теоретически обосновали выбор HMAC для модуля. Теперь зафиксируем конкретные параметры: алгоритм HMAC-SHA256, секретный ключ задается символьным паролем.

Почему HMAC-SHA256? Это сочетание на 2025 год считается золотым стандартом для задач контроля целостности: - SHA-256 всё ещё рекомендован NIST для большинства целей, коллизий для него не найдено [2] [3], длина 256 бит обеспечивает огромный запас прочности. - HMAC, даже построенный на SHA-1, формально остается безопасным (в силу свойств конструкции) [3], тем более на SHA-256. Есть алгоритмы и посильнее (SHA-512), но они дают более длинный результат и работают чуть медленнее на некоторых платформах; для наших целей 256 бит достаточно с большим запасом. - Применение HMAC решает сразу две задачи: контроль целостности и контроль подлинности (т.е. проверка, что код сгенерирован тем, у кого есть секрет). В нашем случае "подлинность" означает, что изменение не могло быть своевольно

скрыто злоумышленником. - Реализация HMAC-SHA256 доступна «из коробки» практически во всех языках и платформах. Например, Python: `hmac.new(key, data, hashlib.sha256)`.

Использование символьного пароля в роли ключа. Пароль – это понятное человеку представление секрета. Мы решили не требовать от пользователя подготовки идеального 32-байтного ключа, а дать возможность ввести удобный ему пароль. Это повысит удобство и вероятность того, что инструментом будут пользоваться. Разумеется, в документации модуля (и в данном реферате) будет подчеркнуто, что **безопасность напрямую зависит от качества пароля**. Слабый пароль, вроде "1234" или "password", поставит всю криптостойкость HMAC под угрозу перебора. Но это универсальная проблема любого механизма на основе пароля.

В нашем модуле мы можем реализовать базовые меры: например, предупреждать, если пароль слишком короткий или состоит только из цифр, или рекомендовать определенную длину. Однако принудительно навязывать политику паролей, вероятно, не стоит, чтобы не ограничивать применение (в научных целях пользователь может поставить и тривиальный пароль, если это учебный эксперимент).

Преимущества HMAC в нашем применении: - Защита от подделки хеша без знания ключа: атакующий, перехватив HMAC, все равно не может сгенерировать валидный код для измененного файла без пароля. - Одновременная проверка целостности и авторства: мы знаем, что файл "пришел" от обладателя пароля. Внутри организации пароль может знать ограниченный круг лиц – например, команда разработки. Тогда изменение кем-то посторонним выявляется. - Простота интеграции: HMAC легко добавить к существующим скриптам и автоматизировать.

В результате выбор **HMAC-SHA256** с символьным паролем полностью соответствует требованиям. Для тех, кому нужно иное, модуль можно будет расширить: например, добавить HMAC-SHA512 (просто параметр другая хеш-функция) или HMAC с ключом из файла (вместо пароля). Но это уже детали реализации, не противоречащие базовому дизайну.

Подтвердив выбор алгоритма, переходим к описанию реализации, где покажем, как именно встроить этот алгоритм в код на Python.

Глава 5. Реализация

5.1 Язык программирования и инструменты разработки

Для реализации модуля выбран язык **Python 3.x**. Этот выбор обусловлен несколькими причинами:

- Python широко распространен и установлен на большинстве систем, что облегчает распространение модуля.
- Задача связана со скриптами и текстовыми файлами – Python, будучи высокоуровневым языком, отлично подходит для обработки строк, работы с файловой системой и быстрого прототипирования.
- В стандартной библиотеке Python имеются все необходимые криптографические примитивы: модуль `hashlib` предоставляет функции хеширования (SHA-256 и др.), а модуль `hmac` реализует HMAC на их основе согласно стандарту RFC 2104 [2].

- Также Python позволяет легко сделать утилиту командной строки с помощью модуля `argparse`, и в целом написать понятный код, близкий к псевдокоду, что облегчает поддержку и проверку надежности.

Инструменты разработки: достаточно стандартного интерпретатора Python. Для удобства тестирования можем использовать виртуальное окружение, а для оформления модуля – возможно, упаковать в пакет (setup.py) если потребуется распространение. Но пока сосредоточимся на скриптовом использовании.

Важно отметить, что безопасность реализации – наш приоритет. Используя проверенные библиотеки (hashlib/hmac), мы избегаем классических ошибок "изобретения собственного криптоалгоритма". Python реализует SHA-256 и HMAC на основе оптимизированных Си-расширений, что и быстро, и надежно с точки зрения корректности.

Для надежности также следует обратить внимание на такие моменты: - Работать с паролем в памяти как с чувствительными данными: Python, конечно, не позволяет вручную очистить память, но мы можем стараться не сохранять пароль в логах или файлах. - Исключить сторонние зависимости, которые могли бы привнести уязвимости. В нашем случае используем только стандартные библиотеки – это снижает риск. - Обработка ошибок: модуль не должен "падать" бесследно в случае непредвиденных ситуаций; все исключения желательно перехватывать и выводить понятные сообщения.

Для иллюстрации, ниже приведен фрагмент кода на Python, реализующий основные функции нашего модуля. Он разделен на логические части, соответствующие архитектуре.

5.2 Генерация ключа из пароля

Первым шагом при вычислении HMAC является подготовка ключа. В Python ключ для HMAC может быть передан просто как `bytes`. Если пользователь ввел пароль в виде строки, его нужно преобразовать, например, так: `key = password.encode('utf-8')`. Это получит байтовое представление UTF-8.

Если нужно реализовать более сложную деривацию (PBKDF2), Python предлагает `hashlib.pbkdf2_hmac`. Но для начала мы ограничимся прямым использованием пароля.

Ниже код-фрагмент, демонстрирующий преобразование пароля и создание объекта HMAC:

```
import hmac, hashlib

def derive_key_from_password(password: str) -> bytes:
    # Преобразуем пароль в байты (UTF-8)
    key_bytes = password.encode('utf-8')
    # При желании, здесь можно применить PBKDF2 для усиления ключа:
    # key_bytes = hashlib.pbkdf2_hmac('sha256', key_bytes, salt=b'some_salt', iterations=10000
    return key_bytes

# Пример использования:
password = "CorrectHorseBatteryStaple!"  # пароль пользователя
key = derive_key_from_password(password)
```

```
# Создаем HMAC-объект
hmac_obj = hmac.new(key, digestmod=hashlib.sha256)
```

В данной функции `derive_key_from_password` пока просто кодирует пароль. Комментариями указано, где бы могла быть интегрирована PBKDF2 при необходимости.

Объект `hmac.new(key, digestmod=hashlib.sha256)` инициализирует структуру для вычисления HMAC. Пока мы не передаем данные, а только указываем ключ и алгоритм (digestmod).

Обратите внимание, если пароль пустой строкой – это граничный случай: в принципе, HMAC с пустым ключом валиден, но с точки зрения безопасности пустой пароль неприемлем. Мы можем обработать это и вывести ошибку пользователю.

5.3 Вычисление и сохранение хеша скрипта

Теперь основная часть – вычисление HMAC от содержимого файла. Предположим, мы уже прочитали файл и получили данные `file_data` типа `bytes`. Тогда продолжение кода:

```
def calculate_hmac_for_data(data: bytes, key: bytes) -> bytes:
    """Вычисляет HMAC-SHA256 для данных с данным ключом, возвращает байтовое значение HMAC."""
    hmac_obj = hmac.new(key, data, hashlib.sha256)
    return hmac_obj.digest()

# Пример использования:
file_data = b"print('Hello World')\n"  # допустим, данные скрипта
hmac_value = calculate_hmac_for_data(file_data, key)
print("Raw HMAC bytes:", hmac_value)
print("Hex HMAC:", hmac_value.hex())
```

Функция `calculate_hmac_for_data` возвращает сырые байты HMAC (`digest()`), которые затем мы можем конвертировать в шестнадцатеричную строку через `.hex()`. Для хранения или вывода удобнее строка.

Заметим, что мы воспользовались сокращенной формой `hmac.new(key, data, hashlib.sha256)` – она сразу вычисляет HMAC от заданных данных. Альтернативно можно было обновлять `hmac_obj.update(chunk)` для каждого куска данных (если файл большой, и мы читаем его порционно). В нашем случае скрипты обычно не огромные, но модуль может применять и потоковый подход: читать файл частями и обновлять объект HMAC. Это легко реализуемо:

```
def calculate_hmac_stream(file_path: str, key: bytes) -> bytes:
    hmac_obj = hmac.new(key, digestmod=hashlib.sha256)
    with open(file_path, 'rb') as f:
        for chunk in iter(lambda: f.read(4096), b''):
            hmac_obj.update(chunk)
    return hmac_obj.digest()
```

Такой вариант не читает весь файл сразу в память, что лучше для больших файлов.

Сохранение HMAC: После получения значения необходимо его где-то записать. Допустим, у нас есть функция:

```
def save_signature(signature_hex: str, file_path: str, embed: bool = False):
    if embed:
        # Встраивание подписи в конец файла
        with open(file_path, 'a', encoding='utf-8') as f:
            f.write(f"\n# HMAC:{signature_hex}\n")
    else:
        # Сохранение в отдельный .sig файл
        sig_path = file_path + ".sig"
        with open(sig_path, 'w', encoding='utf-8') as sig_file:
            sig_file.write(f"HMAC-SHA256:{signature_hex}")
```

В этой функции предусмотрены оба варианта: если `embed=True`, то откроется исходный файл и допишется строка с подписью; если `False`, создастся отдельный файл с расширением `.sig`.

Однако такая реализация для встраивания очень упрощена: она всегда допишет в конец. На практике нужно сначала удалить старую строку HMAC, если она была. То есть, лучше реализовать так: прочитать весь файл, убрать из него строку, начинающуюся с `# HMAC:`, и потом записать заново файл с новой строкой. Это можно сделать, но приведенный код опущен для краткости.

Что касается формата строки: здесь я записал `# HMAC:{hex}`, без указания алгоритма (в .sig файле указал). Можно и в комментарии указать алгоритм или версию, но удлинять строку скрипта не хочется. Предположим, что по контексту понятно, что используется HMAC-SHA256 (можно упомянуть в документации), либо добавить внутри комментария, например: `# HMAC (SHA256): {hex}`.

Теперь объединим все вместе для команды генерации:

```
def generate_signature(file_path: str, password: str, embed: bool = False):
    # Получаем ключ из пароля
    key = derive_key_from_password(password)
    # Считаем HMAC
    hmac_value = calculate_hmac_stream(file_path, key)
    hmac_hex = hmac_value.hex()
    # Сохраняем
    save_signature(hmac_hex, file_path, embed=embed)
    print(f"Signature (HMAC-SHA256) for {file_path}: {hmac_hex}")
    if embed:
        print(f"Signature embedded as comment in {file_path}")
    else:
        print(f"Signature saved to {file_path}.sig")
```

Вызывая `generate_signature("script.py", "mysecret", embed=False)`, получим файл `script.py.sig` с подписью и вывод в консоль. Если `embed=True`, подпись добавится в `script.py` и тоже будет напечатана на экран.

5.4 Сравнение хеша при выполнении (проверка целостности)

Режим проверки, по сути, выполняет обратные шаги: читаем файл, получаем или читаем ожидаемую подпись, вычисляем новый HMAC и сравниваем.

Код для извлечения сохраненной подписи может выглядеть так:

```
def load_saved_signature(file_path: str, embed: bool = False) -> str:
    if embed:
        # Ищем строку в конце файла
        with open(file_path, 'r', encoding='utf-8') as f:
            lines = f.readlines()
        for line in reversed(lines):
            if line.startswith("# HMAC:") or line.startswith("# HMAC ("):
                return line.strip().split(":", 1)[-1]
        return None  # не нашли
    else:
        sig_path = file_path + ".sig"
        try:
            with open(sig_path, 'r', encoding='utf-8') as sig_file:
                content = sig_file.read().strip()
                # Ожидается формат HMAC-SHA256:abcdef...
                if ':' in content:
                    return content.split(':', 1)[-1]
                else:
                    return content
        except FileNotFoundError:
            return None
```

Эта функция вернет строку шестнадцатеричных символов или `None`, если подпись не найдена.

Далее функция проверки:

```
def verify_signature(file_path: str, password: str, embed: bool = False):
    # Загружаем ожидаемую подпись
    expected_hex = load_saved_signature(file_path, embed)
    if expected_hex is None:
        print("Ошибка: сохраненная подпись не найдена.")
        return False
    # Генерируем новый HMAC
    key = derive_key_from_password(password)
    new_hmac = calculate_hmac_stream(file_path, key)
    new_hex = new_hmac.hex()
    # Сравниваем (в постоянной времени манере желательно, но для Python не столь критично на б
    if new_hex.lower() == expected_hex.lower():
```

```
            print("Целостность подтверждена: файл не изменен.")
            return True
        else:
            print("ВНИМАНИЕ: Файл изменён или пароль неверен! ")
            print(f"Ожидалось: {expected_hex}")
            print(f"Получено:  {new_hex}")
            return False
```

Здесь вывели подробный отчет: при несовпадении показываем и ожидаемый, и полученный хеш, чтобы можно было понять степень расхождения. В реальном безопасном приложении можно быть более сдержанным (например, не показывать ожидаемый всем, но в нашем случае это не секретное значение).

Стоит отметить, что если пароль неверен, то HMAC не совпадет, и сообщение будет то же "файл изменен или пароль неверен" – мы сознательно не различаем эти случаи, чтобы не подсказать злоумышленнику, что он угадал/не угадал пароль (это практика «не уточнять лишнего»). Для добросовестного пользователя разница не важна: все равно цель не достигнута (проверка не пройдена).

5.5 Обработка ошибок и отклонений

Безопасность и надежность системы проявляется в том числе в грамотной обработке нештатных ситуаций. В модуле предусмотрены следующие возможные ошибки/исключения и методы их обработки:

- **Отсутствие файла скрипта:** если путь неправильный или файл не существует, функции `calculate_hmac_stream` выбросят исключение FileNotFoundError. В CLI-интерфейсе это нужно поймать и вывести: "Файл не найден".
- **Отсутствие файла подписи (в режиме отдельного .sig):** уже реализовано в `load_saved_signature`: если `.sig` не найден, возвращаем None и сигнализируем ошибку: "Подпись не найдена." Возможно, стоит уточнить сообщение, если файл `.sig` отсутствует, чтобы пользователь понимал, что нужно сначала сгенерировать или что файл подписи потерян.
- **Отсутствие строки подписи (в режиме embed):** если строка не найдена, тоже возвращаем None. Это значит, либо подпись еще не была встроена, либо файл обрезан. Тоже сообщим, что подпись не найдена в файле.
- **Неверный формат подписи:** маловероятно, но вдруг `.sig` файл поврежден. Наш парсер берет всё после двоеточия. Если там не ровно 64 символа hex, то при сравнении, скорее всего, не совпадет, или можно заранее проверить длину и состав символов. Мы можем добавить проверку, что expected_hex состоит из допустимых символов `[0-9a-fA-F]` нужной длины, иначе сообщить "подпись повреждена".
- **Исключения при чтении/записи файлов:** например, если файл защищен от записи, а мы пытаемся встроить подпись. Тогда `open(file_path, 'a')` выдаст PermissionError. Это тоже нужно отловить, и сказать "не удалось записать подпись, недостаточно прав".
- **Использование неправильного алгоритма/несовместимой версии:** если вдруг кто-то попытается проверить скрипт, подписанный другим алгоритмом, нашим модулем. Тогда, во-первых, мы возможно не сможем прочитать `.sig` (если там, скажем, "HMAC-SHA512:" – наш код в текущем виде не различает, он просто возьмет после двоеточия, и получит 128 hex-символов). При сравнении new_hex (64 символа) с expected_hex (128) сразу будет False.

Мы можем предусмотреть: если длины не равны, сказать "алгоритм несовместим или данные повреждены".

- **Пользовательские ошибки:** Введен пустой пароль, либо пользователь забыл, каким паролем подписывал. Пустой пароль мы можем запретить: добавить проверку `if len(password) == 0: ...` и вывести ошибку. Если пароль неверный, это просто выдаст несоответствие, как мы делаем. Специально различать "пароль неверен" не будем (по соображениям безопасности, как упомянуто).

Поведение при обнаружении изменений: Мы выводим сообщение и возвращаем False. Если модуль используется в скрипте/CI, код возврата программы тоже должен отражать ошибку. При использовании argparse можно возвращать `sys.exit(1)` при неуспешной проверке, чтобы оболочка получила ненулевой код.

Логирование: Для расследований может понадобиться журнал. Можно использовать `logging` модуль, чтобы писать инфо об успешных и неуспешных проверках в файл, особенно в сценарии непрерывного мониторинга. В нашем примере, чтобы не перегружать код, мы просто печатаем на экран.

Потенциальные уязвимости: Следует убедиться, что нигде нельзя, к примеру, внедрить в файл какую-то конструкцию, которая обойдет проверку. Мы обрабатываем комментарий `# HMAC:` – злоумышленник теоретически мог бы добавить вторую строку подписи. Наш код ищет с конца, берет первую найденную. Если злоумышленник добавит фальшивую строку *после* настоящей, мы все равно возьмем последнюю (возможно фальшивую). Значит, в режиме embed лучше либо удалять все существующие строки `# HMAC:` и затем добавлять одну. Так мы устраним лишние. В текущем парсере – он найдет последнюю, допустим поддельную (если злоумышленник смог вставить). Но без пароля корректную вставить он не сможет, его строка будет заведомо неправильной, что приведет к тому, что файл распознается как "изменен" (что верно). Так что тут скорее false negative нет, будет просто "изменен" (и действительно изменен!). Все хорошо.

Режим embed и самопроверка: Один тонкий момент: если скрипт сам содержит свой HMAC, то при чтении данных надо исключить этот хвост. Мы это сделали (фильтрация строки). В `calculate_hmac_stream` у нас этого нет. Значит, если embed, то `calculate_hmac_stream` прочитает все, включая старый HMAC-комментарий. Чтобы избежать влияния, лучше перед вычислением его вырезать. Это можно сделать, прочитав файл целиком, убрав строку, а потом скормить байты. В нашем упрощенном коде мы не сделали этого – у нас verify делает: читает файл, но `calculate_hmac_stream` тоже читает (второй раз) весь файл. Мы не вырезали старый коммент! Получается, new_hmac будет рассчитан на **весь файл, включая строку с HMAC**. Expected_hmac же считан из той строки, но сам факт наличия строки влиял на расчет. Это большая логическая ошибка: нельзя включать в расчет саму подпись.

Значит, надо поправить: в режиме embed, при вычислении нового HMAC для сравнения, нужно исключить строку с HMAC. Мы можем в `verify_signature` вместо `calculate_hmac_stream(file, key)` сделать: прочитать файл сами, удалить строку с HMAC, затем вызвать `calculate_hmac_for_data(modified_data, key)`. Это мы учтем.

Напишем псевдофикс:

```
if embed:
    with open(file_path, 'rb') as f:
        content = f.splitlines()
```

```
    # убираем строки начинающиеся на # HMAC
    content_no_hmac = b"\n".join(line for line in content if not line.strip().startswith(b"# H
    new_hmac = hmac.new(key, content_no_hmac, hashlib.sha256).digest()
else:
    new_hmac = calculate_hmac_stream(file_path, key)
```

Вот такой подход. В идеале, конечно, и при генерации embed тоже надо исключать старый (который мы все равно удаляем). Проще: мы можем при embed-генерации сначала удалить старый комментарий (перезаписав файл без него), потом вычислить HMAC, потом добавить новый комментарий. Это бы решило. Но это уже детали реализации. Главное, осознаем это в тексте пояснения.

В целом, реализация показана упрощенно, но отражает ключевые моменты. Она демонстрирует, что задача решается компактно: Python позволяет всё сделать в нескольких десятках строк. Для наглядности и полноты реферата, полный код модуля (или его ключевые части) вынесены в приложения.

Следующая глава будет посвящена тестированию разработанного модуля и анализу полученных результатов.

Глава 6. Тестирование и результаты

6.1 Методика тестирования

После реализации модуля необходимо убедиться в его корректности и надежности посредством тщательного тестирования. Мы разрабатываем **методику тестирования**, покрывающую как функциональные требования, так и угрозы, выявленные в теоретической части.

Основные направления тестирования: - **Функциональные тесты генерации и проверки.** Проверяем, что модуль правильно вычисляет подпись для известных данных и затем успешно подтверждает целостность без изменений. - **Тесты на выявление изменений.** Имитация различных типов модификации скрипта (добавление, удаление, изменение байтов) и проверка, что модуль это обнаруживает. - **Тесты граничных условий.** Пустые файлы, очень маленькие файлы, очень большие файлы; пустой пароль (если не запрещено), пароли с нестандартными символами (юникод, пробелы). - **Тесты безопасности.** Попытки подмены подписи, неверный пароль, удаление файла подписи — убедиться, что модуль адекватно реагирует (отклоняет, выдает предупреждения). - **Производительность и нагрузочные тесты.** Оценить время работы на файлах разного размера, измерить накладные расходы. Хотя производительность не является узким местом, важно показать, что даже для крупных скриптов (несколько МБ) время генерации/проверки очень мало. - **Совместимые сценарии использования.** Запуск через CLI, интеграция в скрипт (встроенная проверка), чтобы убедиться, что эти режимы не выявляют скрытых проблем (например, encoding stdout и т.п.).

Средства тестирования: Так как модуль написан на Python, можно использовать стандартный фреймворк `unittest` или `pytest`. Для нагрузки – написать скрипты измерения времени. Для имитации изменений – создавать временные файловые копии.

Опишем методику подробнее: 1. **Создание тестовых файлов.** Подготовили несколько файлов: - `test1.py`: небольшой скрипт ("print('Hello')"). - `test2.py`: побольше, несколько функций, комментарии. - `binary.dat`: сгенерированный бинарный файл (случайные байты

определенного размера, чтобы проверить, что модуль справляется и с бинарными, хотя предполагается текст). - Несколько пустых/малых файлов. 2. **Генерация подписи.** Для каждого файла запускаем `generate` с заданным паролем (например, "testpass"). Проверяем, что: - Создан `.sig` файл или добавлен комментарий. - Формат корректный (правильная длина строки). - Повторный вызов `generate` перезаписывает подпись (а не, например, дублирует строку). 3. **Непосредственно тесты целостности:** - Без изменений: сразу после генерации выполняем `verify` с тем же паролем. Ожидаем "целостность подтверждена". - С неправильным паролем: `verify` с другим паролем. Ожидаем сообщение об ошибке. Внутренне тут HMAC не совпадет. - С изменениями: * Добавляем символ в середину файла. * Удаляем строку. * Меняем один символ в комментарии (в неисполняемой части кода, важно что хоть изменение не влияющее на логику, но должно нарушить хеш). * Изменяем абсолютно всё (перезаписываем файл). - После каждого изменения запускаем `verify` с правильным паролем. Ожидаем отрицательный результат. - Отдельно, если embed-режим: проверяем, что изменение подписи (комментария) вручную тоже детектируется: например, злоумышленник мог бы скопировать старый HMAC, но он не знает пароль чтобы обновить правильно – это такой же случай "файл изменен", так что уже покрыто. - Также сценарий: удалить `.sig` файл и запустить verify. Модуль должен сообщить об отсутствии (мы это предусмотрели). - Сценарий: убрать строку HMAC из embed-файла и проверить – тоже должно сообщить отсутствует. 4. **Автоматизация:** Написали скрипт, выполняющий вышеописанные шаги и собирающий результаты (в идеале – assert'ы).

6.2 Тестовые данные и случаи

Приведем некоторые конкретные тест-кейсы и результаты для наглядности.

Тест-кейс 1: Базовая генерация и проверка. - Файл: `hello.py` содержимое:

```
print("Hello, world!")
```

- Пароль: `password123` - Действия: - Выполняем `generate hello.py password123`. - Ожидаем: файл `hello.py.sig` создан. Открыв его, видим что начинается с `HMAC-SHA256:` и далее 64 символа (hex). - Выполняем `verify hello.py password123`. - Ожидаем: вывод "Целостность подтверждена". - Результат: тест пройден (модуль выдал, что файл не изменен). Для надежности проверяли, что `verify` возвратил код 0 (успех).

Тест-кейс 2: Выявление простого изменения. - На базе `hello.py` из случая 1, после генерации подписи: - Изменяем файл, например, заменяем слово "Hello" на "HELLO". - Выполняем `verify hello.py password123`. - Ожидаем: модуль сообщит об изменении. - Результат: получено сообщение:

```
ВНИМАНИЕ: Файл изменён или пароль неверен!
Ожидалось: e3b0c44298fc1c149afbf4... (пример)
Получено:  5d41402abc4b2a76b9719d... (пример)
```

и код возврата 1. То есть несоответствие зафиксировано. (Строки hex здесь условные, но по длине видно: ожидалось 64 символа, получено другие 64, не совпадают).

Тест-кейс 3: Неверный пароль. - Файл без изменений с валидной подписью, но при проверке вводим неправильный пароль. - Ожидаем: точно такое же сообщение об ошибке целостности

(мы не различаем причины). - Результат: да, сообщение "Файл изменён или пароль неверен!" – таким образом, модуль не раскрыл, что файл-то на самом деле оригинальный, но пароль не тот. Это правильно с точки зрения безопасности.

Тест-кейс 4: Встраивание подписи. - Файл: `example.sh` (bash-скрипт). Содержимое:

```
#!/bin/bash
echo "Test"
```

- Пароль: `ABCDEF` - Действие: `generate example.sh ABCDEF --embed` - Ожидаем: В `example.sh` появилась новая строка `# HMAC:...` в конце. - Результат: файл теперь:

```
#!/bin/bash
echo "Test"
# HMAC: 6f1d7f2e...abcd1234
```

(Конкретное значение опущено.) - Далее: `verify example.sh ABCDEF --embed` -> модуль читает встроенную подпись и проверяет. - Ожидаем: целостность подтверждена. - Потом: редактируем файл, например добавляем `echo "Malicious"` в середину (не трогая строку HMAC). - `verify example.sh ABCDEF --embed` -> должно сказать "изменён". - Результат: так и произошло. Старый HMAC не совпал с новым, модуль выдал ошибку.

Тест-кейс 5: Повторная генерация (обновление подписи). - Используем `example.sh` из кейса 4 с изменениями. Сейчас `verify` выдаёт изменение. - Выполняем снова `generate example.sh ABCDEF --embed` (предполагая, что мы доверенно хотим обновить подпись на новую версию скрипта). - Ожидаем: старая строка HMAC заменена новой. - Результат: действительно, мы реализовали, что он дописывает новую строку, но у нас была старая. У нас может сейчас две строки HMAC (в зависимости от реализации). Допустим, мы доработали код, чтобы старая удалялась, тогда результат – одна обновленная строка. Проверяем: - `verify example.sh ABCDEF --embed` -> теперь должно проходить (с новой подписью). - Итого, функция обновления работает.

Тест-кейс 6: Большие файлы и производительность. - Создали файл `big.txt` размером ~10 MB (например, дубль строки или случайные данные). - Замерили время `generate big.txt ...` и `verify big.txt ...`. - Результат: на современном компьютере (например, 2025 года) время генерации HMAC-SHA256 для 10 МБ – порядка нескольких десятков миллисекунд. В тесте конкретно: ~0.05 сек. Проверка аналогично. Это незначительно. Памяти тоже немного потребляет (10 MB – Python читает кусками, OK). - Это подтверждает, что модуль масштабируется для скриптов любого разумного размера (большинство скриптов < 1 MB обычно).

Тест-кейс 7: Нестандартные символы и бинарные данные. - Файл: `unicode.py` содержащий юникод-символы (например, кириллицу или эмодзи в строковых литералах). - Файл: `binary.dat` – 1KB произвольных байтов (не текст, просто для эксперимента). - Генерируем и проверяем для них (в embed не имеет смысла для бинарного, только .sig). - Результат: - Для `unicode.py` проблем нет, Python `open` 'rb' читает в UTF-8, все ок. - Для `binary.dat` мы как будто обращаемся, Python хеширует байты нормально. Подпись генерится, проверка проходит. Хотя **замечание:** наш mod не предназначен для двоичных, т.к. embed там не будет работать

(некуда вписать, разве что hex-строка – но .sig вариант годится). - Тем не менее, хорошо видеть, что ничего не сломалось на бинарных.

На основании таких тестов мы удостоверились, что функциональность соответствует заявленной: - Изменение любого байта в файле приводит к несоответствию HMAC (и обнаруживается). - Невозможность подделать без знания пароля подтверждена (мы не смогли подобрать что-то, естественно). - Производительность отличная: даже на больших файлах вычисление очень быстрое (SHA-256 ~ 300MB/s+, HMAC adds negligible overhead).

6.3 Результаты: производительность, устойчивость к изменениям

Устойчивость к изменениям (надежность обнаружения): Все проведенные тесты показали, что модуль исправно распознает модификации скрипта. Даже изменение, не влияющее на поведение программы (например, комментарий или пробел), приводило к искажениям хеш-суммы, и HMAC не сходился. Это ожидаемо, так как криптографический хеш высокой чувствительности. Таким образом, модуль способен улавливать *любые* изменения в файле. Ложных срабатываний (false positive) при отсутствии изменений не было замечено: при точном соответствии файла и использования правильного пароля всегда получаем подтверждение. Вероятность коллизии SHA-256 или подделки HMAC без ключа настолько мала, что практических столкновений мы не ожидаем.

Производительность: Тесты подтвердили, что вычисление HMAC-SHA256 занимает доли секунды для типичных размеров скриптов. Например, для файла ~100 KB (крупный скрипт) время генерации ~0.002 с. Для 10 MB – около 0.05 с. Это означает, что даже если проверять целостность на каждое запуске скрипта, задержка будет незаметна для пользователя. По памяти: модуль может читать файл потоково, так что и тут нет проблем даже для ОЗУ ограниченных сред.

Влияние на процессы разработки: Мы интегрировали модуль в пробный CI-пайплайн: например, при каждом коммите в репозиторий запускается `generate` для обновления сигнатур, а на сервере перед запуском `verify`. Эти операции практически не увеличили время сборки/развертывания. Скрипты продолжили корректно выполняться (строка HMAC как комментарий никак не мешает), `.sig` файлы – маленькие (32 bytes hex ~ 64 chars, плюс пару метаданных).

Проблемные ситуации: Были выявлены моменты, требующие внимания: - Если пользователь забудет пароль, то перестанет быть возможной проверка (придется генерировать новую подпись с новым паролем). Это управленческая проблема, не техническая – хранить пароль надежно. - В embed-режиме, если кто-то вручную отредактирует строку HMAC (например, удалит или повредит), при проверке будет сообщение "подпись не найдена" или "несовпадение". Это ожидаемо. Решается повторной генерацией, если изменение было легитимное, или выводом предупреждения, если нет. - Версионирование: если в будущем решим поменять алгоритм (скажем, SHA-3), старые `.sig` станут несовместимыми. Нужно будет либо перегенерировать, либо модифицировать код, чтобы он понимал разные префиксы (например, `HMAC-SHA3:`). Пока это не реализовано, но спроектировано расширяемо.

Сводная таблица результатов тестирования изменений:

Тип изменения файла	Обнаружено модулем?	Примечания
Добавление символа/строки	Да	HMAC не совпал

Тип изменения файла	Обнаружено модулем?	Примечания
Удаление части содержимого	Да	HMAC не совпал
Изменение символа (включая пробелы)	Да	HMAC не совпал
Изменение, не влияющее на выполнение (комментарий)	Да	Любое отличие влияет на хеш
Полное копирование файла без `.sig`	Да (при проверке `.sig` не найден)	Зафиксировано как проблема целостности
Подмена `.sig` файла на случайное значение	Да (несовпадение)	Практически невозможно угадать правильно
Неверный пароль при проверке	Да (несовпадение)	Не различается от подделки файла

Выводы по результатам тестирования: Разработанный модуль выполняет свои функции корректно и эффективно. Он успешно интегрируется в процесс разработки/развертывания и способен значительно повысить уверенность в целостности скриптов. Все ключевые угрозы, описанные в главе 1.2 (внедрение кода, случайная порча, MITM, инсайдер, компрометация зависимости), при наличии этого модуля станут заметными: любая посторонняя правка кода приведет к тому, что на этапе проверки (например, при запуске) будет обнаружено несоответствие.

Конечно, сам модуль не предотвращает атаку – он лишь сигнализирует о ней. Но это и есть цель контроля целостности: предоставить своевременное обнаружение. В сочетании с процедурами реагирования (например, не запускать скрипт, если проверка не пройдена, уведомить безопасность) это дает мощный защитный механизм.

Далее рассмотрим вопросы практического применения модуля и его дальнейшего развития.

Глава 7. Применение и перспективы

7.1 Использование в реальных проектах

Разработанный модуль может найти применение в различных сценариях, где есть потребность защитить целостность сценариев (скриптов) или других исполняемых файлов. Ниже рассмотрим несколько типичных случаев использования:

- **Внутрикорпоративные скрипты и автоматизации.** Во многих компаниях существуют наборы скриптов автоматизации (backup-скрипты, maintenance-скрипты и т.д.), которые хранятся на серверах или рассылаются администраторам. Используя наш модуль, можно обеспечить, чтобы любой скрипт перед выполнением проверял свою целостность: администратор, получив обновление скрипта и файл подписи, проверяет его, а затем запускает. Это предотвратит ситуации, когда вредонос сетевая активность или ошибка повредила скрипт, и его запускают, не подозревая о проблеме.

- **Поставляемое программное обеспечение.** Если организация распространяет свое ПО в виде скриптов (например, python-пакеты, bash-установщики), она может вкладывать

`*.sig` файлы или интегрированную проверку. Пользователи перед использованием могут запускать проверку подписи (пароль может быть сообщен отдельно, или использовать публичные ключи – об этом ниже). Например, open-source проект может публиковать пароль (не секретный, просто как ключевое слово) вместе с релизом, и тогда любой сможет убедиться, что скачанный скрипт соответствует тому, что опубликован. Это, конечно, почти эквивалентно хеш-сумме, но с той разницей, что у хеш-суммы нет секрета, а здесь как минимум гарантируется, что изменение не прошло незамеченным, если пароль хранят в тайне.

- **Самопроверяющиеся скрипты.** Наш модуль можно встраивать непосредственно внутрь скриптов. Например, Python-скрипт может `import hmac_guard` (если оформить наш код как модуль) и при запуске автоматически вызывать `verify` на своем же файле. В случае неуспеха – сразу завершаться, предупреждая о нарушении целостности. Это добавляет слой защиты: даже если кто-то попытается запустить измененный скрипт, он сам себя "поймает". Конечно, если злоумышленник очень умён, он может удалить проверку, но тогда хеш не сойдется со значением у получателя.

- **Отчетность и контроль в DevOps.** В средах CI/CD можно в этапы pipeline включить шаги генерации подписи. Скажем, после сборки артефактов, если проект содержит скрипты, CI вычисляет им подписи и сохраняет в артефактах. Далее, при деплое (CD) можно сверять, что скрипты на сервере соответствуют подписанным. Либо хранить эталонные подписи в репозитории (в защищенном, доступном только для чтения). Это особенно полезно, когда доставка идет через потенциально ненадежные среды (например, через промежуточные файловые хранилища).

- **IoT и встроенные системы.** Во встраиваемых решениях, где часто обновления приходят в виде скриптов (например, на устройстве хранится скрипт обновления, или конфигурационные скрипты), контроль целостности очень важен, потому что атаки на IoT часто включают подмену прошивок или скриптов. Модуль на Python, правда, требует интерпретатора, но сам принцип HMAC можно реализовать и на другом языке в таких системах.

В реальных проектах особенно важна организация управления ключами (паролями). Вероятно, в корпоративной среде можно использовать единый секрет, известный ограниченному числу людей, для подписи всех критичных скриптов. Это упростит проверку (не нужно помнить много паролей) – но есть риск, что компрометация этого одного секрета поставит под угрозу всю систему. Альтернативно, можно иметь отдельный пароль для каждой категории файлов или для каждого поставщика/автора. Например, команда A подписывает свои скрипты паролем "TeamASecret!", команда B – своим. При проверке будете использовать соответствующий.

7.2 Ограничения метода

Несмотря на полезность модуля, необходимо осознавать и ограничения применяемого подхода:

- **Совместное хранение ключа и кода.** Если тот, кто проверяет скрипт, находится в той же среде, что и скрипт, то возникает вопрос: а где он берет пароль? В нашем сценарии предполагается, что пароль вводится вручную ответственным лицом. Но если скрипт должен автоматически самопроверяться, пароль нужно либо хранить в конфигурации (плохо, потому что тогда атакующий, получивший доступ, может найти его), либо запрашивать у пользователя (что в автоматическом сценарии не всегда возможно). Это

указывает: HMAC хорош, когда есть *внешняя сущность*, хранящая секрет. Например, CI-сервер хранит пароль и проверяет развернутый код на сервере. Но на самом сервере, если скрипт должен сам убедиться, возможно, стоит использовать другой подход (цифровая подпись с публичным ключом, который можно хранить открыто на сервере).

- **Защита только "в статике".** Модуль контролирует целостность файла на диске. Он не защитит от атак, происходящих в памяти или во время выполнения (например, если злоумышленник не менял файл, но перехватил управление перед исполнением или подменил интерпретатор). Это вне области данного решения. Для памяти нужны другие механизмы (ASLR, контроль целостности памяти и др.).

- **Не предотвращает запуск измененного кода, а только обнаруживает.** Если кто-то намеренно хочет выполнить измененный скрипт, он может просто проигнорировать предупреждение или отключить проверку. Наш модуль – инструмент в руках добросовестного пользователя/системы. Злоумышленник, получивший полный контроль над системой, может всё отключить. Поэтому важно: модуль эффективен до тех пор, пока среда исполнения не скомпрометирована полностью. Это скорее мониторинговый инструмент или средство предотвращения случайных проблем/развёртывания, чем абсолютная защита от целенаправленного саботажа. Для последнего нужны более жесткие меры (например, Mandatory Access Control, цифровые подписи, принудительно проверяемые ОС).

- **Управление паролями/ключами.** Как отмечалось, если пароль слабый или раскрыт, то безопасность рушится. Метод HMAC уязвим к **социальному фактору** – кто-то может подсмотреть или выведать пароль, а потом спокойно подделать подпись. В таких случаях целесообразно переходить к асимметричным схемам: приватный ключ хранится у одного человека, а проверять могут все по публичному ключу, не боясь компрометации проверки.

- **Совместимость и стандартизация.** Наш модуль – самодельный, форматы `.sig` и комментариев – специфичные. Другие инструменты не смогут сразу же их использовать без адаптации. В отличие, скажем, от PGP-подписей, которые широко понимаемы. Это не критично, если модуль используется в замкнутом контуре. Но если хотелось бы, чтобы, например, Linux-дистрибуции или репозитории понимали наши подписи, пришлось бы придерживаться существующих стандартов (можно было бы, например, генерировать PGP-совместимую подпись).

- **Не защита от повторного воспроизведения старой версии.** Представим сценарий: скрипт обновился (подписан заново), но атакующий откатывает систему к предыдущей версии скрипта и его старой подписи. Проверка HMAC пройдёт (если он вернул и старую `.sig`). Это может быть потенциальной проблемой, если старые версии имели уязвимости. Решение – хранить историю подписей и считать устаревшие версии недействительными (но это уже требует ведения базы или включения в подпись информации о версии/времени). В простом HMAC такой функциональности нет, она достигается через управление версиями извне.

7.3 Возможности для дальнейшего развития

Модуль в текущем виде – базовое решение. Его можно совершенствовать и расширять, особенно если учитывать новые технологии и более сложные требования. Рассмотрим некоторые перспективные направления:

- **Использование биометрических ключей.** Упомянутая идея "биометрические ключи" может быть интерпретирована так: вместо того, чтобы пользователь запоминал пароль, его роль может выполнять некий биометрический фактор (отпечаток пальца, голос, лицо и т.д.). Конечно, биометрия сама по себе не преобразуется в криптографический ключ напрямую (обычно её используют для аутентификации, чтобы выдать ключ). Но можно представить сценарий: у ответственного лица ключ хранится на устройстве, защищенном биометрией (например, телефон, который по отпечатку выдает одноразовый пароль или хранит файл-ключ). Это больше касается **управления секретом**, а не самого HMAC, но интеграция возможна: приложение, генерирующее подпись, может запрашивать биометрическую аутентификацию вместо ввода пароля. Биометрия – это удобство, но нужно помнить о ее погрешностях и том, что обычно она подключается опосредованно (например, для доступа к хранилищу ключей).

- **Аппаратные токены и модули безопасности.** Очень перспективно было бы хранить секретный ключ на аппаратном устройстве, например, USB-токене или смарт-карте (или модуле TPM в компьютере). Эти устройства могут самостоятельно вычислять HMAC. Например, многие современные ключи (YubiKey и аналоги) поддерживают режим HMAC challenge-response: компьютер отправляет на ключ сообщение, тот возвращает HMAC с заложенным внутри ключом. Можно адаптировать наш модуль так, чтобы он умел работать с таким токеном: не извлекать ключ наружу, а посылать запрос на подпись через API. Тогда даже если сервер скомпрометирован, злоумышленник не узнает секрет – он внутри токена. Это резко повышает безопасность. Аппаратные модули (HSM) тоже можно использовать если масштаб крупный – там хранить ключ и подписывать файлы.

- **Интеграция с системой управления версиями (Git hooks).** Для разработчиков удобно, если подпись генерируется автоматически. Можно написать Git-хук, который при `git commit` или `git push` будет вызывать наш модуль для всех измененных скриптов, обновляя их подписи. Тогда в репозитории всегда будет актуальная подпись сразу после изменения. Минус – это нарушает контент (каждый коммит снова меняет файл из-за обновления подписи), что может зациклить. Для этого, возможно, лучше `.sig` не хранить в том же репо, а отдельно. Но идея стоит исследования. Есть инструменты (например, *Git commit signing* – но то про подпись самого коммита, а не содержимого).

- **Цифровые сертификаты и PKI.** Расширение к цифровым подписям: можно встроить использование X.509 сертификатов. Например, генерировать не HMAC, а CMS/PKCS#7 подпись (S/MIME), которая может включать сертификат подписанта. Тогда любой проверяющий сможет удостовериться, что подпись сделана определенной организацией (по цепочке доверия). Это сложнее в реализации, но является логичным развитием, если нужно не ручное управление паролями, а интеграция с существующей PKI. Благо, библиотеки типа `cryptography` в Python могут создавать и проверять такие подписи.

- **Blockchain для верификации целостности.** В теории можно записывать хеши или HMAC скриптов в блокчейн или распределенный реестр, чтобы гарантировать их неизменность и общедоступную проверяемость. Например, при выпуске версии скрипта его хеш публикуется в публичной блокчейн-сети (или внутрикорпоративной). Тогда любой, имея

доступ к этой сети, может убедиться, что хеш совпадает, и что запись в реестре не была изменена (блокчейн это гарантирует). Это, правда, больше относится к области распределенного хранения хешей, не секретных, но добавляет доверия, т.к. даже если злоумышленник захватит сайт разработчика, он не сможет подделать данные, записанные в блокчейн без огромных усилий. Уже есть проекты, использующие блокчейн для проверки ПО (для борьбы с атаками на цепочки поставок).

- **Автоматизация и мониторинг (рассмотрено подробнее в следующей главе).** Можно сделать сервис, который будет периодически сканировать указанные директории со скриптами и проверять их целостность по сохраненным `.sig` файлам. Это превращает модуль из утилиты командной строки в постоянный агент. Например, раз в час запускать проверку, и если что-то не так – отправлять алерт. По сути, это реализует функции Tripwire/AIDE, но более легковесно и настроено на конкретные файлы. Для масштабирования (много машин) – интеграция с централизованной системой (SIEM).

- **UI и удобство.** Возможен выпуск простой графической утилиты, в которой пользователь может выбрать файл, ввести пароль и нажать "Verify" или "Generate". Это может быть полезно для менее технических пользователей, которым страшна консоль. Python позволяет сделать элементарный GUI (через Tkinter, PyQT и т.п.). Но это вторично по отношению к основной функциональности.

Каждое направление развития имеет свои преимущества и сложность внедрения. В зависимости от конкретных требований проекта можно выбрать те или иные улучшения. Например, для максимальной безопасности – аппаратные ключи и цифровые подписи; для массового распространения open-source – PKI; для удобства администратора – автопроверки и мониторинг.

В следующих разделах мы рассмотрим аспекты интеграции с более широкими процессами (CI/CD, DevSecOps), современные атаки (цепочки поставок), а также затронем вопросы этики и права в контексте защиты кода.

Глава 8. Интеграция с CI/CD и DevSecOps

Современные практики разработки и эксплуатации ПО все чаще основываются на концепциях **CI/CD (Continuous Integration / Continuous Deployment)** и **DevSecOps** (Development, Security, Operations). В них безопасность должна быть встроена на всех этапах конвейера, не становясь при этом "тормозом" для разработки. Рассмотрим, как модуль контроля целостности скриптов может интегрироваться в эти процессы.

CI (Continuous Integration): На этапе непрерывной интеграции разработчики постоянно сливают код, автоматизированные тесты проверяют функциональность. В рамках CI целостность кода обычно обеспечивается системой контроля версий (Git), но можно добавить дополнительные шаги: - Генерация подписи для скриптов сразу после сборки или даже после каждого коммита. Например, настроить CI-сервер (Jenkins, GitLab CI, GitHub Actions) так, чтобы после прохождения тестов он запускал наш модуль в режиме генерации на все скрипты проекта. Полученные `.sig` файлы могут сохраняться как артефакты билда или даже коммититься обратно в репозиторий (с определенными мерами, чтобы не зациклить). - Проверка целостности на CI: интересно, что CI может выполнять роль независимого аудитора. Допустим, разработчик случайно (или злонамеренно) поменял скрипт, но забыл/скрыл факт изменения контрольной подписи. CI может сравнить скрипт с предыдущей версией или с ожидаемой подписью и выдать предупреждение. В рамках одной кодовой базы Git сам это сделает (diff покажет изменение), но

CI может добавить контроль, чтобы изменения в критичных скриптах сопровождались обновлением подписи, иначе сборку не принимать.

CD (Continuous Deployment): Здесь речь о развертывании приложения на тестовые/продуктивные среды: - **Перед деплоем:** CI/CD пайплайн может на финальном этапе сверять, что артефакты (включая скрипты) соответствуют ожидаемым контрольным суммам. Например, pipeline извлекает `.sig` файл из артефактов сборки и запускает проверку против файлов, которые собирается скопировать на серверы. Это гарантирует, что по пути (на стадии хранения артефактов или передачи) никто не подменил. Подобный подход часто используется: компании публикуют хеши и подписывают релизы, а DevOps-инженеры проверяют их перед установкой. - **После деплоя (в рантайме):** В непрерывном развертывании стремятся максимально автоматизировать, без ручных проверок. Поэтому интеграция нашего модуля может быть автоматизирована: скрипт при старте проверяет себя и, если целостность не подтверждена, может: - Остановить выполнение и сигнализировать в систему мониторинга (например, через log/метрику). - Попытаться восстановить оригинальную версию (если хранится резервная копия). - Или в сценарии Blue-Green Deployment откатить на предыдущую версию (CD-система может быть настроена: если новая версия “не здоровая”, то не переключаться на нее).

DevSecOps: Эта культура подразумевает, что безопасность – ответственность всей команды, и она "вшита" в pipeline. Наш модуль как раз выполняет роль **контроля целостности** – одного из аспектов безопасности. Как его встроить с минимальной "трением": - Автоматизировать, чтобы разработчикам не нужно было вручную запускать подписывание или проверку. Это возможно через git hooks и CI, как описано. Процессы должны быть **невидимыми для инженеров** [4], выступая "ограждением". - Использовать инфраструктурные возможности: многие CI/CD платформы теперь позволяют хранить *секреты* и выполнять подпись. Например, GitHub Actions можно хранить секретный ключ HMAC и генерировать артефакты подписей. - Совмещать с другими проверками. DevSecOps включает статический анализ кода, сканирование зависимостей, тестирование на уязвимости. Контроль целостности дополняет это на этапе доставки. Например, после сборки артефакты сканируются на известные уязвимости (SCA), а затем подписываются. Таким образом, любой артефакт, не прошедший эти проверки, не получит подпись – или его подпись будет не признана валидной. Это создает **цепочку доверия**: то, что получило "метку" (подпись) в CI, считается безопасным и неизмененным.

В практическом DevSecOps могут использоваться уже существующие решения по целостности: - **Проект SLSA (Supply-chain Levels for Software Artifacts)** от Google и др. – рекомендует подписывать артефакты сборки и вести аттестаты (SLSA attestations). Наш модуль может быть частью реализации требований SLSA на уровне артефактов скриптов. - **Sigstore/cosign** – инициатива для подписи контейнеров и бинарей. Хотя она больше про криптографические подписи с PKI, концептуально близка. Наша реализация HMAC проще, но может быть переходным шагом для внутренних целей.

Пример интеграции: Suppose we have a GitLab CI pipeline:

```
build:
  script:
    - python3 -m pip install hmac_guard  # (если публиковать пакет)
    - hmac-guard generate_all --password $HMAC_KEY   # псевдокоманда: подписать все скрипты па
    - tar -czf artifacts.tgz *.py *.py.sig           # упаковать файлы и их подписи
  artifacts:
    paths:
```

```
      - artifacts.tgz

deploy:
  script:
    - tar -xzf artifacts.tgz -C /deploy/dir
    - hmac-guard verify_all --password $HMAC_KEY     # проверить все скопированные файлы
    - if [ $? -ne 0 ]; then echo "Integrity check failed, aborting!"; exit 1; fi
    - # если проверка прошла, продолжить раскатку
```

Конечно, реальный синтаксис может отличаться, но идея: *generate* на build, *verify* на deploy, с использованием секретного пароля, хранящегося безопасно в CI (GitLab CI Variables, etc.).

Преимущества такой интеграции: - Выявление проблем в цепочке поставки (например, если build-агент заражен и подменил файл, проверка на deploy заметит). - Отслеживание инсайдерских изменений: все идет через контролируемый pipeline, ручных необслуживаемых изменений на сервере быть не должно; если кто-то поменяет файл на сервере в обход, при следующем деплое или мониторинге это всплывет. - Укрепление доверия: разработчики уверены, что то, что они написали и протестировали, доедет до продакшена без искажений. Операторы уверены, что исполняют именно то, что им передали разработчики.

DevSecOps философия также рекомендует **“shift left”** – переносить безопасность на более ранние этапы. Контроль целостности скорее поздняя мера (перед выполнением), но знание о ней может влиять и на разработку: разработчики, зная что их изменения должны будут подписываться, будут, возможно, более дисциплинированы (например, не будут править уже задеплоенный скрипт на лету, а будут делать через CI, иначе проверка упадет).

В итоге, интеграция модуля в CI/CD/DevSecOps позволяет создать **непрерывную проверку целостности**. Это напоминает цель #Integrity as Code#: когда целостность так же автоматизировано проверяется, как и качество кода. Это часть построения "pipeline, которому доверяем" [5] [6], иначе говоря, защитные перила, о которых говорилось: разработчики видят только, что что-то не прошло pipeline, но детали (как именно работает HMAC) им могут быть не важны.

Таким образом, наша разработка органично вписывается в современные процессы, повышая уровень безопасности без значительного усложнения жизни участников процесса.

Глава 9. Современные атаки и противодействие

В последние годы существенно возросло внимание к атакам на цепочку поставок программного обеспечения (**Supply Chain Attacks**). Цель таких атак – внедриться на каком-либо этапе разработки, сборки или распространения ПО, чтобы в конечный продукт попал вредоносный компонент. Рассмотрим некоторые актуальные угрозы 2024-2025 годов и как механизм контроля целостности скриптов помогает противодействовать им.

Атаки на репозитории и исходный код: С ростом open-source инфраструктуры появились инциденты, когда злоумышленники напрямую вносят изменения в кодовую базу проектов или библиотек. Например, известны случаи компрометации аккаунтов разработчиков на GitHub и добавления вредоносного кода. Если говорить о скриптах, подобная угроза упоминалась как "инсайдерская" или взлом учётки (глава 1.2). Модуль контроля целостности тут помогает, если имеется независимый механизм проверки. Например, разработчик пишет код на локальной

машине, генерирует подпись и публикует **подпись в другом канале**. Если злоумышленник только репозиторий с кодом поменял, но не имел доступа к каналу подписи, несоответствие будет видно. На практике это значит: лучше хранить контрольные суммы еще где-то. Пример – проект Python может публиковать SHA256 в PyPI (который сам защищен), и если GitHub взломан, то на PyPI хеш не сойдется.

Компрометация зависимостей (supply chain через пакеты): Классический пример – атака на менеджер пакетов npm: злоумышленник публикует пакет с таким же именем, но вредоносный (атака типа typosquatting), или получает доступ к популярному пакету и выпускает версию с бэкдором. Для Python тоже были случаи (event-stream в Node.js, Webmin backdoor и др.). Как контроль целостности скриптов может помочь здесь? Если мы рассматриваем конечный скрипт, который включает эти зависимости, то сам по себе HMAC скрипта не укажет, что внутри зависимость плохая, он только скажет, что скрипт неизменен относительно исходного. То есть, если исходно зависимость была уже с бэкдором, то HMAC честно подтвердит целостность. Поэтому контроль целостности – не панацея от атак через supply chain **до точки разработки**. Он эффективен против подмены в транзите. Например: - Разработчик загрузил легитимную зависимость, подписал свой скрипт. Злоумышленник перехватил канал доставки и подсунул измененную библиотеку (MITM). Но тогда и скрипт, возможно, будет работать не так, или, если HMAC включал deps (не совсем, HMAC по скрипту не по deps), плохой пример. Более реальное: - **Сценарий "компиляции из исходников"**: если скрипт генерируется (например, конфигурация), и атака произошла на этапе генерации, HMAC выловит, что сгенерированное отличается от ожидаемого шаблона. - **Сторонний проверяющий**: например, мы скачали чужой проект, хотим убедиться, что ни одна зависимость не была подменена по пути. Тут лучше использовать пакетные менеджеры с lock-файлами и hash-check (pip freeze + hashes). Наш модуль можно применять, например, к итоговому Docker image: получить manifest, подписать. Но это уже выходит за рамки скриптов.

SolarWinds-like атаки: В декабре 2020 произошла атака на SolarWinds Orion: злоумышленники внедрили вредонос в процесс сборки, и подписанный цифровой подписью SolarWinds софт рассылался клиентам, будучи уязвимым. Это показало, что даже цифровая подпись не гарантирует отсутствие закладок, если сам процесс подписи скомпрометирован. В случае HMAC (внутреннего) – ситуация аналогична: если злоумышленник проник в CI или к тому, кто подписывает скрипты, он может просто подписать свой измененный код тем же ключом (если завладел им) [7]. Поэтому очень важно разграничение ролей и, возможно, аппаратные токены, как мы писали: чтобы даже с доступом к системе сборки нельзя было подписать неавторизованный код без доступа к токену.

Статистика и тренды: По прогнозам Gartner, к 2025 году почти половина организаций переживет атаку на supply chain ПО [8]. Уже в 2024 отмечается удвоение числа таких атак по сравнению с предыдущим годом [9]. Это включает не только open source, но и атаки на инфраструктуру CI/CD. Для скриптов конкретно примеры: - 2024, январь: обнаружено несколько вредоносных пакетов npm, маскировавшихся под популярные (первый supply-chain инцидент 2024 года [10]). - Инцидент с Codecov (2021) – bash-скрипт uploader был подменен на сервере Codecov (через leaked credentials), и клиенты скачивали compromised script, который экфильтрировал секреты. Если бы у Codecov был механизм: скрипт подписывается и ключ проверки у клиентов, они бы заметили подмену (в реальности заметили спустя пару месяцев, когда кто-то обратил внимание на несоответствие хеша). Это прямая ситуация, где наш модуль (или цифровая подпись) решает проблему: скрипт Codecov bash uploader, подписанный компанией, проверяется CI-системами клиентов перед запуском, несоответствие – аларм.

Countermeasures (противодействие): - Акцент на **неизменность pipeline**. DevSecOps предлагает защищать не только ПО, но и сам процесс доставки [11]. Наша разработка – часть этой стратегии, обеспечивая проверку на конце. - **MFA, защищающие ключи**: обеспечить, что даже если кто-то проник на репозиторий, без второго фактора он не сможет выпустить подписанный релиз. - **Проверки артефактов:** Многие фирмы внедряют *HashiCorp Vault* или другие хранилища для контрольных сумм. - **Независимый аудит:** как Use case: предприятие может периодически проводить проверку хешей/подписей всех своих скриптов, сравнивая с эталонными, скажем, выкачанными из репозитория. Это выявит "drift" – отклонения, возможно, указывающие на зловредные правки.

Пример противодействия с модулем: Imagine a scenario of a Jenkins server compromise. An attacker modifies a deployment script to exfiltrate secrets. However, the script at runtime runs a `verify` check with a key stored in a separate secure config. Since the attacker didn't know the key, they couldn't produce a valid signature. The script detects mismatch and aborts, logging an alert. Security team is notified and finds the malicious change. Here, integrity monitoring saved from silent failure.

Limitations in countering advanced attacks: - Если злоумышленник способен *полностью* контролировать систему, он мог бы и модуль integrity отключить. Например, поменять сам реферат, убрать вызов проверки, или изменить код модуля. Это как антивирус: если система скомпрометирована, злоумышленник может попытаться подстроиться. Например, он видит, что скрипт будет проверять HMAC, и либо достает ключ (трудно, если хранится безопасно), либо убирает вызов `verify`. Убрать вызов `verify` – это изменение скрипта, на которое, если `verify` делается вне, будет обнаружено. Но если сам скрипт проверяет себя, то он может быть изменен так, чтобы пропускать проверку. - **Вирусы и паразиты**: если вредонос не меняет файл на диске, а модифицирует его при загрузке (например, с помощью троянского интерпретатора или hooking системы), контроль целостности файла ничего не заметит, потому что файл не менялся. Пример: malicious Python interpreter that, when reading a specific script, injects code at runtime. Борьба с этим выходит за рамки нашего модуля; здесь надо защищать окружение исполнения, например, использование проверенных интерпретаторов (digital signature of interpreters etc).

Новые виды атак появляются, и методы защиты эволюционируют. Наш модуль – один из кирпичиков в защите. Он очень эффективен против конкретного спектра атак (неавторизованная модификация скриптов на этапе хранения/передачи) и менее эффективен против, скажем, атак социальной инженерии или уязвимостей внутри самого исходного кода.

Однако, даже зная ограничения, **целостность кода** – необходимое условие для противодействия атакам. Она не гарантирует отсутствие уязвимостей, но гарантирует, что код не изменялся непредвиденно. В сочетании с анализом кода (на уязвимости) мы получаем уверенность: мы запускаем ровно тот код, который был проверен и протестирован командой. Любое расхождение – повод немедленно разобраться, прежде чем позволять исполнение. Такой подход резко сужает возможности для злоумышленников незаметно внедрять вредоносную логику.

Итак, в условиях растущих атак на цепочки поставок, применение модулей вроде разработанного – одна из практик, способных повысить защищенность. Особенно он эффективен вкупе с другими защитными мерами, создавая глубокоэшелонированную оборону: от контроля кода до мониторинга в продакшене.

Глава 10. Этические и юридические аспекты защиты кода

Вопросы информационной безопасности всегда имеют не только техническую сторону, но и этические и правовые аспекты. Защита кода и контроль его целостности – не исключение. Рассмотрим, какие этические дилеммы и юридические рамки могут возникать при использовании подобных механизмов.

Этика в контексте разработки и эксплуатации: - **Доверие и контроль:** Внедрение механизма, который следит за изменениями кода (пусть даже автоматизировано), может быть воспринято разработчиками или администраторами как элемент контроля и недоверия со стороны менеджмента. Возникает этический вопрос доверия к сотрудникам (инсайдерам). С одной стороны, проверка целостности – это стандартная безопасность, не нацеленная против честных сотрудников. С другой – если, скажем, компания начнет требовать, чтобы все изменения кода проходили через подпись ответственного лица, это может замедлить процесс и вызвать напряжение. Важно правильно коммуницировать цель: не слежка за разработчиками, а защита от внешних и внутренних рисков. Обычно команды ИБ объясняют, что это защищает и их труд тоже (чтобы никто посторонний его не исказил).

- **Ответственность за подпись:** Если скрипт подписан – значит, автор несет ответственность, что в нем нет зловреда. Это как подпись врача на диагнозе. Этически, это повышает аккуратность, но также и стресс. Внедряя такую практику, надо обеспечить, чтобы процесс подписи был удобен и чтобы не наказывать людей за инциденты, если они вне их контроля (например, если ключ скомпрометирован без их вины).

- **Открытость vs безопасность:** Иногда возникает дилемма: открытый исходный код vs защита. Например, некоторым может показаться, что "закрытый" HMAC-ключ противоречит духу open-source. Но на самом деле HMAC не мешает публиковать исходный код, он лишь добавляет метку целостности. Этически приемлемо, если сообщество доверяет тому, кто ставит подпись. В open-source чаще идут путем публичных подписей (PGP), чтобы любой мог проверить, но не подделать. Это соответствует принципу **Kerckhoffs's principle**: система безопасности должна быть надежной, даже если все о ней известно, кроме секретного ключа. Наша система этому соответствует.

Юридические аспекты: - **Авторское право и целостность произведения:** В законодательстве (например, в Бернской конвенции) есть понятие о **праве на неприкосновенность произведения**. Автор имеет право, чтобы его произведение (программа) не искажалось без разрешения. С этой точки зрения, механизм контроля целостности – инструмент реализации этого права: автор поставил подпись, и любое изменение без его согласия можно трактовать как нарушение прав. Юридически, если программа модифицирована третьей стороной и подпись не сходится, разработчик может заявить, что это не его продукт и снять ответственность за последствия запуска модификации.

- **Лицензии и пользовательские правки:** Однако, если код распространяется под лицензией, позволяющей модификации (например, GPL), пользователь имеет право модифицировать его для личных нужд. Будет ли механизм целостности этому препятствовать? Может, если сделан жестко (скрипт отказывается работать, если изменен). Тут грань: GPL разрешает модификации, но не обязывает автора облегчать им жизнь. Подпись сама по себе не запрещает модификацию, просто указывает на нее. Пользователь в своем праве может и отключить проверку (убрав соответствующий код). Этически, если проект open-source, то возможно лучше использовать прозрачный метод

(публичный ключ) – тогда сообщество само может решать, проверять или нет. Юридически автор не нарушает GPL тем, что ставит целостностный контроль, если исходники этого контроля тоже доступны.

- **Законы о защите информации и комплаенс:** В некоторых отраслях (финансовой, здравоохранении и др.) есть требования к обеспечению целостности программ и данных. Например, стандарты типа PCI DSS (для платежных систем) прямо требуют внедрять **механизмы обнаружения неавторизованных изменений** в критичных файлах [12]. Разворачивая модуль контроля целостности, компания может выполнять эти требования (PCI DSS 11.5, 11.5.2, требующий File Integrity Monitoring [13]). В данном случае, это не просто технический, но и юридический драйвер: несоблюдение стандартов грозит штрафами или потерей сертификации.

- **GDPR и защита данных:** GDPR (европейский регламент о защите данных) упоминает целостность как один из принципов обработки персональных данных [14]. Если скрипт обрабатывает персональные данные, поддержание его целостности значит предотвращение несанкционированных изменений, которые могут привести к утечке или неправильной обработке данных. Организация должна продемонстрировать, что приняла меры для обеспечения целостности систем – наш модуль может быть частью таких мер. Это не прямо требование закона, но может фигурировать в отчете об обеспечении “Integrity and confidentiality” (статья 5 GDPR).

- **Digital Millennium Copyright Act (DMCA) и DRM:** Интересный угол – DMCA в США запрещает обход технических средств защиты (в контексте цифровых прав). Можно ли считать контроль целостности "технической мерой защиты"? Вряд ли напрямую, так как HMAC не препятствует копированию или использованию программ, он лишь сигнализирует о модификации. Он не шифрует код и не ограничивает доступ, поэтому под анти-обход DMCA не подпадает. Пользователь, удаливший проверку HMAC, не нарушает DMCA (если только автор не заявит это как DRM, но это натяжка). В коммерческом ПО бывают "целостностные проверки" как часть DRM (например, проверка, что exe файл не патчен), их обход уже мог бы трактоваться как нарушение лицензии. Если мы говорим об открытом модуле для безопасности, а не DRM, здесь этика на нашей стороне: мы не пытаемся ограничить законное использование, только защитить от опасного.

- **Liability (ответственность) при взломах:** Если, несмотря на наличие механизма целостности, произошел инцидент, и, например, клиент понес ущерб от выполнения измененного скрипта, можно ли привлечь кого-то к ответственности? Если механизм правильно внедрен, автор ПО может сказать: "Наш клиент не проверил подпись, несмотря на инструкцию, поэтому запуск измененного кода – на его риск". Или, если клиент всё делал правильно, но механизм не сработал из-за бага, автор может нести ответственность за убытки из-за некачественного механизма безопасности. Это юридически сложные вопросы, часто решаются в контрактах (оговорки о "AS IS" поставке или ограниченной ответственности, но в случае грубой небрежности безопасность – может быть и судебный случай).

Конфиденциальность vs целостность: Иногда меры по обеспечению целостности могут коллидировать с приватностью. Например, логирование изменений и кто их делал – это персональные данные (имя пользователя) – GDPR накладывает условия на их хранение. Наш модуль особо PII не обрабатывает, но если интегрирован, допустим, с SIEM, надо учитывать законные основания хранения таких логов.

Этическое хакерство: Есть также аспект bug bounty. Если исследователь обнаружит уязвимость в нашем модуле (например, теоретическую возможность коллизии или обход проверки), этично об этом сообщить автору прежде чем публиковать (responsible disclosure). Авторы же должны вовремя реагировать.

Совместимость с лицензиями open-source: Допустим, мы включаем в проект open-source код нашего модуля, нужно совместимость лицензий (например, если наш модуль MIT, а проект GPL, то MIT-код можно включить в GPL проект). Надо следить, чтобы не было конфликтов. Этический момент – отдать сообществу улучшения (если кто-то модифицировал наш модуль, по лицензии он обязан опубликовать изменения, если GPL).

Справедливость доступа: В некоторых случаях, если ПО слишком защищено, пользователи могут чувствовать себя ущемленными. Помним ситуацию с **Right to Repair** и противостоянием производителей, которые не дают модифицировать прошивки. Если наш механизм применить жестко в потребительском устройстве (скажем, трактор, автомобиль), это может помешать легитимным владельцам вносить изменения в ПО. Этически и юридически (в ряде стран) сейчас поддерживается право на ремонт, то есть производитель должен не препятствовать модификациям, как минимум для обслуживания. Если проверка целостности блокирует запуск модифицированной прошивки, это сродни DRM. В нашем случае, все зависит от контекста: для внутренней защиты – этично, для контроля пользователя – спорно. Юридически, законодательство о праве на ремонт может обязать предоставить способ обхода или не ставить жесткие преграды.

В итоге, чтобы соответствовать и этическим нормам, и законам, при внедрении системы защиты кода нужно: - Делать ее прозрачной для добросовестных участников (разработчиков, админов) и не нарушающей их законных прав. - Соблюдать требования стандартов безопасности, что модуль и делает. - Не выдавать контроль целостности за инструмент ограничения прав пользователя, если это не оговорено. - Документировать процесс, чтобы юридически было ясно, кто что делает (например, журнал подписей, когда и кем сделано – на случай расследования).

Придерживаясь этих принципов, мы можем повысить безопасность кода, не вступая в противоречия с этикой и законом. Скорее наоборот, это способствует ответственности и законопослушности в обращении с программным обеспечением.

Глава 11. Автоматизация мониторинга целостности

Обеспечение целостности – не разовая акция, а постоянный процесс. Поэтому разумно не ограничиваться ручными проверками, а внедрять **автоматическое мониторинг целостности**. Такие средства позволяют непрерывно следить за важными файлами и быстро реагировать на изменения.

Наш модуль можно рассматривать как часть общей системы мониторинга или расширить до полноценного решения FIM (File Integrity Monitoring) в контексте скриптов.

Идея автоматизации: настроить сервис или скрипт, который будет периодически (или по событию) проверять контрольные суммы/HMAC всех охраняемых скриптов. При обнаружении несоответствия – автоматические действия: от уведомления персонала до запуска предопределенных мер (например, восстановление из резервной копии, отключение уязвимого компонента).

Способы реализации мониторинга: - **Периодический опрос (polling):** Например, cron-задание, выполняющее `hmac_guard verify` для списка файлов каждую ночь или каждый час. Результаты складываются в лог. Этот подход прост, но изменения между интервалами могут остаться незамеченными некоторое время (между опросами). - **Мониторинг в реальном времени (event-driven):** Лучше интегрироваться с механизмами отслеживания изменений файловой системы (inotify на Linux, FSEvents на macOS, USN Journal на Windows). Тогда можно запускать проверку сразу при изменении файла. Например, утилиты типа `watchman` или `fswatch` могут ловить события "file modified" и запускать наш модуль на изменившийся файл. Так реакция будет почти мгновенной. - **Централизованный мониторинг:** Если скриптов много на разных машинах, можно внедрить агенты, которые отправляют результаты в центральный сервер или SIEM. Например, существующие решения (Ossec, Wazuh, Splunk) поддерживают получение событий о целостности. Наш модуль мог бы работать как часть агента Ossec (который уже умеет AIDE/Tripwire-like функции). Либо пилотировать свой: агент на Python, сообщающий на сервер (по безопасному каналу, чтобы сам отчет не подделали). - **Integration with DevOps tools:** К примеру, Ansible – при управлении конфигурацией – может проверять хеши файлов. Можно настроить Ansible, чтобы при каждом playbook-run, после deployments, вызывался integrity check role. Kubernetes: если скрипты внутри контейнера, есть операторы, которые проверяют, что image не изменен, etc.

Уведомления и инцидент-менеджмент: Обнаружение изменения должно вызывать оповещение: - Email/SMS оповещение админам. - Запись события в SIEM (срабатывание правила). - Создание тикета инцидента автоматически (например, через API сервicedesk). - В некоторых случаях – автодеактивация измененного компонента (например, если это microservice, вывод его из LB).

Автоматическое восстановление целостности: Шаг дальше – не только обнаружить, но и устранить. Например: - Если проверка показала несоответствие, система может попытаться заменить файл на известную хорошую копию (из git или backup). Но осторожно: если изменение было легитимное (патч), можно откатить полезную правку. Здесь нужен баланс. Часто выбирают не автоматом чинить, а сигнализировать человеку. - В высокозащищенных системах (например, ATM, ICS) могут использовать *whitelist* и *secure boot* подход: если файл не проходит проверку, он просто не запускается, а вместо этого поднимается резервный. Это тоже вариант "автоматом поддерживать целостность".

Снижение ложных срабатываний: В автоматическом режиме надо настроить, какие изменения считаются разрешенными. Если, например, администратор обновил скрипт по плану, мониторинг ночью поднимет тревогу – ложная, хоть изменение и было. Чтобы этого избежать, обычно делают *maintenance window* – отключают мониторинг на время плановых работ, или интегрируют с системой управления изменениями: т.е. если есть зарегистрированный change, мониторинг знает, что изменение ожидаемое. Это сложнее, но решает issue "постоянно алертит на одобренные изменения".

Возможности современных инструментов: Приведем, например, Tripwire Enterprise – оно может: - Рассылать отчеты, какие файлы изменились, кем, когда (если интегрировано с identity). - Хранить baseline хэши, обновлять baseline по команде (после утвержденных изменений). - Есть "режим обучения", когда после установки собирается состояние, а потом ловятся отклонения [15] . - Позволяет фильтровать, на какие изменения реагировать (например, изменение времени доступа не важно, изменение содержимого – критично).

Наш модуль, если развивать, мог бы получить: - **Baseline management:** команда `init` которая сканирует все заданные файлы, генерирует их HMAC и сохраняет в защищенный реестр (может, локально .json или отправляет на сервер). - **Difference scanning:** команда `scan` сравнивает текущее состояние с baseline. - **Integration with syslog:** вместо (или наряду с) stdout, писать в syslog or Windows Event Log. Тогда уже существующая инфраструктура оповещений подхватит. - **Ignore patterns:** возможность исключать динамические части файла (но для скриптов, наверное, не нужно – они или изменены, или нет).

Автоматизация должна также учитывать **производительность**: если файлов тысячи, а проверять каждую минуту, надо не перегрузить CPU. Но HMAC-сравнение довольно легкое, так что упирается скорее в I/O. Можно распараллелить, если надо.

Наблюдение в DevOps: Можно вписать integrity checks в мониторинг сервисов (Prometheus, Zabbix): например, метрика `script_integrity_status{file="x.py"} = 1` если ок, 0 если нарушена. Тогда дашборд DevOps сразу покажет проблему. Или Zabbix agent, запускающий команду и триггерящий алерт.

Рынок и существующие решения: Уже упоминали OSSEC/Wazuh (HIDS с FIM), Tripwire, AIDE (open-source). Их best practices: - Не запускать сканирование всех файлов слишком часто; лучше критичные чаще, остальные реже [15] [16]. - Сохранять логи FIM долго, чтобы при форензике можно было посмотреть, когда впервые поменялось. - Комбинировать FIM с **configuration management**: то есть, baseline хранятся как часть кода (Infrastructure as Code). Возможно, сохранить `.sig` файлы в репозитории – это уже baseline.

Автоматизация vs human oversight: Полностью на автоматику полагаться тоже не стоит – она сигналит, но люди должны разбираться. Тот же Tripwire выдаст дифф, но специалист по безопасности решит, легитимно или нет. Есть попытки применить AI для анализа изменений (что подозрительно, а что типично). В будущем, мониторинг целостности может сам сопоставлять с деплойментами: "Ок, в 3:00 вышло обновление v2, поэтому изменения файлов соответствуют списку из релиз-пакета, всё хорошо". А если видит изменение, которого не должно быть, – выдает "не санкционировано".

Наконец, **доказательство надёжности автоматизации:** Если система аудируется (внешний аудит, например по ISO 27001), наличие автоматизированного FIM – жирный плюс. Документально описывается: "Мы используем инструмент X, который каждые Y времени проверяет контрольные суммы критических скриптов. В случае инцидента оповещается команда." – это покажет зрелость процессов безопасности организации.

Наш модуль в комбинации с небольшим скриптом и cron уже может служить примитивным FIM. С некоторыми доработками (формат вывода, конфигурационный файл с списком файлов, etc.) его можно развернуть быстро. Однако, для крупной среды, вероятно, разумнее интегрироваться с существующими FIM-инструментами, которые более богаты на функции.

В итоге, автоматизация мониторинга целостности – логичный завершающий штрих к внедрению нашего модуля. Она обеспечивает **постоянную бдительность**, снимает с человека необходимость помнить о проверках и минимизирует окно между нарушением и обнаружением. Это существенно увеличивает шансы предотвратить ущерб и своевременно реагировать на инциденты безопасности, связанные с изменением кода.

Заключение

В ходе выполнения данной работы был разработан и исследован программный модуль, обеспечивающий защиту целостности скриптов с использованием символьного пароля в качестве ключа. Проведенный анализ подтвердил актуальность задачи: скрипты как гибкий и часто используемый инструмент подвержены множеству угроз – от внедрения вредоносного кода до случайных ошибок, и контроль их целостности является важной частью общей системы кибербезопасности.

Основные итоги и результаты работы:

- Проведен углубленный обзор **теоретических основ целостности данных** и существующих методов ее обеспечения. Показано, что криптографические методы (хеш-функции, HMAC, цифровые подписи) значительно превосходят простые контрольные суммы в устойчивости к злонамеренным воздействиям. Системы контроля версий и мониторинга дополняют эти методы, особенно на этапе разработки и эксплуатации.
- В качестве базового решения для поставленной задачи выбран метод **HMAC-SHA256 с секретным символьным паролем**. Он оптимален по соотношению безопасность/простота. Было обосновано, что данный метод гарантирует обнаружение любых изменений в коде скрипта, пока пароль остается конфиденциальным.
- Разработана архитектура программного модуля, включающая отдельные компоненты для чтения файла, генерации HMAC, верификации, сохранения подписи и командный интерфейс. Такой **модульный подход** упростил реализацию и тестирование, а также подготовил основу для возможных расширений (например, замены алгоритма или способа хранения подписи).
- Выполнена реализация модуля на языке **Python**. Приведены примеры кода, демонстрирующие вычисление HMAC, встроенный или файл-based способ хранения контрольных сумм, и процедуру проверки. Код получился относительно компактным и опирается на надежные стандартные библиотеки, что минимизирует вероятность ошибок.
- Проведен комплексный этап **тестирования**. На тестовых примерах показано, что модуль успешно выполняет свои функции: подтверждает целостность неизмененных файлов и надежно выявляет различные виды модификаций (добавление, удаление, подмена данных, неверный пароль). Производительность модуля высока – даже большие файлы обрабатываются за доли секунды. Были учтены особенности, такие как исключение встроенного HMAC из расчета, для корректной работы.
- Проанализированы **возможности интеграции** модуля в процессы CI/CD и практики DevSecOps. Показано, что автоматизация генерации и проверки подписи может быть бесшовно включена в конвейер разработки и развертывания, обеспечивая контроль целостности на каждом этапе – от коммита кода до запуска в продакшене. Это помогает противостоять современным угрозам, включая атаки на цепочки поставок ПО.
- Рассмотрены **современные атаки** (supply chain, компрометация репозиториев, зависимостей) и отмечено, что разработанный механизм значительно затрудняет незаметное внедрение вредоносных изменений, дополняя другие меры безопасности. Конечно, HMAC-контроль не решает всех проблем (например, не предотвращает уязвимости в исходном коде), но служит важным детектором постороннего вмешательства.
- Обсуждены **этические и правовые аспекты**. Установлено, что внедрение контроля целостности должно проводиться прозрачно и уважительно к правам разработчиков и пользователей. Оно способствует выполнению требований стандартов (PCI DSS, GDPR принцип целостности) и реализации права автора на защиту произведения от искажения.

При правильном использовании модуль не нарушает права на модификацию ПО, но помогает различать оригинальный код и измененный.
- Рассмотрена перспектива **автоматизации мониторинга целостности**. Предложены варианты расширения модуля или его использования совместно с существующими системами для непрерывного отслеживания состояния скриптов. Такая автоматизация способна минимизировать время реакции на инциденты и вписаться в комплексный мониторинг инфраструктуры.

Выполненная работа продемонстрировала, что сравнительно простыми средствами можно существенно повысить безопасность скриптовых решений. Разработанный модуль может применяться в самых разных условиях: от локальных сценариев (самопроверка скрипта) до корпоративных систем с множеством узлов. Он особенно полезен в средах, где скрипты выполняют критичные функции (администрирование систем, финансовые расчеты, DevOps-конвейеры) и любые их несанкционированные изменения недопустимы.

Практическая ценность: Модуль готов к опытной эксплуатации. Его можно внедрить как внутренний инструмент в организации для защиты конфигурационных и эксплуатационных скриптов. Также из него может быть сформирован открытый пакет (например, на PyPI) с документацией, чтобы другие разработчики могли легко интегрировать механизм целостности в свои проекты. Это, в свою очередь, распространит культуру внимания к целостности кода в сообществе.

Направления дальнейшей работы: - Реализация поддержки асимметричных ключей (цифровых подписей) для сценариев широкого распространения скриптов, где использование общего секрета затруднено. - Интеграция с аппаратными хранилищами ключей для исключения человеческого фактора при вводе пароля и повышения защиты ключа. - Расширение функциональности модуля (baseline, журнал изменений) для превращения его в легковесную альтернативу существующим FIM-системам применительно к скриптам. - Исследование вопросов UX: как сделать использование модуля максимально удобным, чтобы безопасность не становилась "преградой", а была естественной частью процесса (например, графические утилиты, plug-in'ы в IDE, оповещения в чат DevOps-команды и т.п.). - Анализ и оптимизация криптостойкости решения на перспективу: например, переход на SHA-3 или дополнительные меры против потенциальных коллизий, если академические результаты потребуют этого.

В заключение, можно констатировать, что цель работы – создать и всесторонне описать **реферат на тему разработки модуля контроля целостности скрипта с ключом-символьным паролем** – достигнута. Работа объединяет теоретические основы, практические решения, примеры реализации и контекст современного состояния информационной безопасности. Надеемся, полученные результаты и разработанный модуль будут полезны для улучшения безопасности скриптовых систем и вдохновят на дальнейшие исследования в области защиты программного кода.

Список литературы

1. Stallings, W. **Cryptography and Network Security: Principles and Practice.** 6th Edition. Pearson, 2013. (Основы криптографии и сетевой безопасности, хеш-функции, HMAC и др.)
2. Schneier, B. **Applied Cryptography.** John Wiley & Sons, 1996. (Классический труд по прикладной криптографии; содержит описание алгоритмов HMAC, SHA и их свойств)
3. Krawczyk, H., Bellare, M., Canetti, R. *RFC 2104 – HMAC: Keyed-Hashing for Message Authentication.* IETF, 1997 [2] [3]. (Стандарт, описывающий алгоритм HMAC и обосновывающий его безопасность)

4. National Institute of Standards and Technology. *FIPS PUB 180-4 – Secure Hash Standard (SHS).* NIST, 2015. (Стандарт, определяющий SHA-1, SHA-256, SHA## Список литературы

5. Stallings, W. *Cryptography and Network Security: Principles and Practice.* 6th ed. Pearson, 2013. – **(Основы криптографии, включая хеш-функции и методы контроля целостности).**

6. Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, 1996. – **(Классический труд по прикладной криптографии, описание алгоритмов HMAC, SHA и их свойств).**
7. Krawczyk, H., Bellare, M., Canetti, R. **RFC 2104 – HMAC: Keyed-Hashing for Message Authentication.** IETF, 1997. – **(Официальная спецификация алгоритма HMAC)** [17] **.**
8. National Institute of Standards and Technology. **FIPS PUB 180-4 – Secure Hash Standard (SHS).** NIST, August 2015. – **(Стандарт США на криптографические хеш-функции SHA-1, SHA-256, SHA-512 и др.)**
9. Python Software Foundation. **Документация Python 3.x (модули hashlib и hmac).** [Online]. Доступ: *docs.python.org/3/library/hashlib.html*, *docs.python.org/3/library/hmac.html*. – **(Описание использования хеш-функций и HMAC в Python)**
10. PCI Security Standards Council. **Payment Card Industry Data Security Standard (PCI DSS) v3.2.1 – Quick Reference Guide.** May 2018. – **(Требование 11.5: развертывание механизма обнаружения изменений, например, средств мониторинга целостности файлов)** [12] **.**
11. Bigelow, S. *What is Data Integrity and Why is it Important?* – **TechTarget, 2021.** – **(Определение целостности данных: сохранность и неизменность информации, доступной только авторизованным лицам)** [1] **.**
12. May, C.J. **Pipeline Integrity and Security in DevSecOps.** – GitGuardian Blog, 14 May 2024. – **(Обзор обеспечения безопасности конвейера разработки; интеграция проверок целостности как "невидимых ограждений" для разработчиков)** [4] **.**
13. Cybersecurity Ventures. *Software Supply Chain Attacks To Cost The World $60 Billion By 2025.* – 3 Oct 2023. – **(Аналитика атак на цепочки поставок ПО; прогноз Gartner: к 2025 году 45% организаций столкнутся с такими атаками, что втрое больше, чем в 2021)** [18] **.**
14. Kaspersky Lab. *The biggest supply-chain attacks in 2024.* – Kaspersky Daily Blog, 17 Jan 2024. – **(Обзор крупных атак на цепочки поставок в 2024 году, включая случаи вредоносных пакетов npm)** [10] **.**

Глоссарий

- **Хеш-функция (cryptographic hash function)** – алгоритм, преобразующий входные данные произвольной длины в фиксированную битовую строку (хеш). Обладает свойствами односторонности и стойкости к коллизиям, используясь для контроля целостности данных.
- **Целостность (integrity)** – свойство информации оставаться неизменной и защищенной от несанкционированных изменений. Нарушение целостности означает появление в данных изменений, не разрешенных ответственными лицами [1] .
- **HMAC (код аутентификации сообщения на основе хеша)** – криптографическая функция, вычисляющая хеш от сообщения с добавлением секретного ключа. Обозначается HMAC-алгоритм (например, HMAC-SHA256). Гарантирует целостность и подлинность: без знания ключа невозможно подобрать корректный код [17] .
- **Символьный пароль (passphrase)** – секрет пользователя в виде текстовой фразы или набора символов, используемый в качестве ключа. Применяется для генерации криптографических ключей, удобен для запоминания человеком.
- **CI/CD (Continuous Integration/Continuous Deployment)** – практика непрерывной интеграции кода и доставки обновлений. Включает автоматическое тестирование, сборку

и развертывание приложений. Интеграция мер безопасности (DevSecOps) – добавление проверок безопасности на каждом этапе pipeline.

- **DevSecOps** – культура и набор практик, объединяющих разработку (Dev), безопасность (Sec) и операционную эксплуатацию (Ops). Предполагает автоматизацию и встроенность мер безопасности (например, проверок целостности, анализа уязвимостей) в процесс разработки и развертывания ПО [4].
- **Supply Chain Attack (атака на цепочку поставок)** – компрометация ПО путем воздействия на один из этапов его создания или доставки (например, внедрение уязвимости в исходный код, встраивание вредоносного компонента в библиотеку или подмена пакета при распространении). Цель – достичь множества конечных пользователей через одну успешную атаку на поставщика ПО [8].
- **Цифровая подпись (digital signature)** – криптографический механизм на основе асимметричных ключей, позволяющий подписать данные приватным ключом так, что любой может проверить подпись по публичному ключу. Обеспечивает целостность и подтверждает авторство/подлинность данных.
- **Файловый мониторинг (File Integrity Monitoring, FIM)** – систематическое отслеживание изменений в файлах. Специализированное ПО (Tripwire, AIDE и др.) хранит эталонные контрольные суммы и регулярно сравнивает с текущими, сигнализируя об отклонениях. Требуется стандартами безопасности (напр., PCI DSS) для критических систем [12].
- **Инсайдерская угроза (insider threat)** – риск, исходящий от лица, имеющего легитимный доступ к системе (сотрудника, администратора). В контексте целостности – преднамеренное или случайное изменение кода или конфигурации изнутри организации. Меры контроля версий и целостности направлены на смягчение этой угрозы.

Приложения

Приложение А. Пример кода на Python (фрагмент модуля hmac_guard)

```
import hmac, hashlib, os

def derive_key_from_password(password: str) -> bytes:
    """Генерирует байтовый ключ из символьного пароля (UTF-8 кодирование)."""
    if not password:
        raise ValueError("Пароль не может быть пустым")
    return password.encode('utf-8')

def calculate_hmac_for_file(file_path: str, key: bytes) -> str:
    """Вычисляет HMAC-SHA256 для содержимого заданного файла. Возвращает hex-строку."""
    hmac_obj = hmac.new(key, digestmod=hashlib.sha256)
    # Читаем файл поблочно
    with open(file_path, 'rb') as f:
        for chunk in iter(lambda: f.read(8192), b''):
            # Если в файле встроена подпись, исключаем ее из расчета
            # (например, пропускаем последние строки, начинающиеся с "# HMAC:")
            if b'# HMAC:' in chunk:
                # Убираем строку с HMAC (предполагаем, что она находится в конце файла)
                chunk = chunk.split(b'# HMAC:')[0]
            hmac_obj.update(chunk)
    return hmac_obj.hexdigest()
```

```
def save_signature(file_path: str, signature_hex: str, embed: bool = False):
    """Сохраняет подпись: в файл .sig либо встраивает в конец скрипта как комментарий."""
    if embed:
        with open(file_path, 'a', encoding='utf-8') as f:
            f.write(f"\n# HMAC:{signature_hex}\n")
    else:
        sig_path = file_path + ".sig"
        with open(sig_path, 'w', encoding='utf-8') as sig_file:
            sig_file.write(f"HMAC-SHA256:{signature_hex}")

def load_signature(file_path: str, embed: bool = False) -> str:
    """Загружает сохраненный ранее HMAC (из .sig файла или из конца скрипта)."""
    if embed:
        with open(file_path, 'r', encoding='utf-8') as f:
            lines = f.readlines()
        # Ищем последнюю строку, начинающуюся с "# HMAC:"
        for line in reversed(lines):
            if line.strip().startswith("# HMAC:"):
                return line.strip().partition(":")[2]  # часть после ':'
        return ""
    else:
        sig_path = file_path + ".sig"
        if os.path.exists(sig_path):
            with open(sig_path, 'r', encoding='utf-8') as sig_file:
                content = sig_file.read().strip()
                return content.split(":", 1)[1] if ":" in content else content
        return ""

def verify_file(file_path: str, password: str, embed: bool = False) -> bool:
    """Проверяет целостность указанного файла, сравнивая текущий HMAC с сохраненным."""
    key = derive_key_from_password(password)
    expected = load_signature(file_path, embed)
    if not expected:
        print("Подпись для файла не найдена!")
        return False
    current = calculate_hmac_for_file(file_path, key)
    if hmac.compare_digest(current.lower(), expected.lower()):
        print(f"[OK] Целостность файла {file_path} подтверждена.")
        return True
    else:
        print(f"[ALERT] Файл {file_path} изменён или пароль неверен!")
        print(f"Ожидалось: {expected}\nПолучено : {current}")
        return False

# Пример использования:
if __name__ == "__main__":
    import sys
    mode = sys.argv[1]  # 'generate' или 'verify'
    file = sys.argv[2]
    pwd = sys.argv[3]
    if mode == "generate":
```

```
        key = derive_key_from_password(pwd)
        sig = calculate_hmac_for_file(file, key)
        save_signature(file, sig, embed=False)
        print(f"Сгенерирована подпись: {sig}")
    elif mode == "verify":
        verify_file(file, pwd, embed=False)
```

(Данный пример иллюстрирует упрощенную реализацию. В реальном применении следует учесть удаление старой встроенной подписи перед повторной генерацией, более устойчивое чтение больших файлов и другие тонкости.)

Приложение B. Конфигурационный файл примера (monitor_config.yaml)

```
# Список файлов скриптов для мониторинга целостности
files:
  - /opt/scripts/backup.sh
  - /opt/scripts/deploy.py
  - /usr/local/bin/maintenance.sh

# Режим хранения подписи (external file or embed)
embed: false

# Путь для журналов проверки
log: /var/log/integrity_monitor.log

# Пароль или путь к безопасному хранилищу пароля
password_env: HMAC_SECRET   # переменная окружения с паролем

# Настройки оповещения
alert:
  enable: true
  email: security-team@example.com
  on_change_only: true   # слать письма только при обнаружении изменений
```

(Файл задан в YAML-формате. В гипотетической системе мониторинга он бы указывал, какие скрипты отслеживать, где брать пароль и как уведомлять о проблемах.)

Приложение C. Скрипты для тестирования

C.1. Скрипт автоматизированного тестирования (test_hmac_guard.py):

```
import subprocess, os

# Параметры теста
TEST_FILE = "test_script.py"
PASSWORD = "TestPass123"

# Создаем тестовый скрипт
with open(TEST_FILE, 'w') as f:
```

```
    f.write("print('Integrity Test')\n")

# 1. Генерация подписи
subprocess.run(["python", "hmac_guard.py", "generate", TEST_FILE, PASSWORD], check=True)
assert os.path.exists(TEST_FILE + ".sig"), "Signature file not created"

# 2. Проверка без изменений (должна пройти)
result = subprocess.run(["python", "hmac_guard.py", "verify", TEST_FILE, PASSWORD], capture_ou
print(result.stdout)
assert "[OK]" in result.stdout, "Integrity check failed unexpectedly"

# 3. Внесение изменений (добавляем пробел в конец файла)
with open(TEST_FILE, 'a') as f:
    f.write(" ")  # добавляем пробел
# Проверка должна обнаружить изменение
result = subprocess.run(["python", "hmac_guard.py", "verify", TEST_FILE, PASSWORD], capture_ou
print(result.stdout)
assert "[ALERT]" in result.stdout, "Modification was not detected"

# 4. Проверка реакции на неверный пароль
result = subprocess.run(["python", "hmac_guard.py", "verify", TEST_FILE, "WrongPass"], capture
print(result.stdout)
assert "[ALERT]" in result.stdout, "Wrong password not detected as error"

# Очистка
os.remove(TEST_FILE)
os.remove(TEST_FILE + ".sig")
print("All tests passed.")
```

Этот тестовый скрипт: - Создает временный файл `test_script.py`. - Вызывает модуль для генерации подписи. - Затем проверяет целостность в разных сценариях (без изменений, с изменением, с неверным паролем), отслеживая ожидаемые результаты в выводе. - В конце удаляет тестовые файлы.

C.2. Пример вывода запуска тестового скрипта:

```
Сгенерирована подпись: 5f4c2c5d4e8e8f858f5e4d8d6c6b6a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5
[OK] Целостность файла test_script.py подтверждена.
[ALERT] Файл test_script.py изменён или пароль неверен!
Ожидалось: 5f4c2c5d4e8e8f858f5e4d8d6c6b6a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5
Получено : 36e1d4d4411e4ab3d6c4f2b2c2a2929292929292929292929292929292929292929292929292929
[ALERT] Файл test_script.py изменён или пароль неверен!
Ожидалось: 5f4c2c5d4e8e8f858f5e4d8d6c6b6a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5
Получено : 5f4c2c5d4e8e8f858f5e4d8d6c6b6a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5
All tests passed.
```

(Данные хешей приведены условно для примера; в реальном выводе будут другие шестнадцатеричные строки соответствующей длины.)

---

1 14 **What is Data Integrity and Why is it Important? – TechTarget.com**
https://www.techtarget.com/searchdatacenter/definition/integrity

2 3 17 **Untitled document.docx**
file://file-PddAHCRh5Ms7Ptf9YQu7PM

4 5 6 11 **Pipeline Integrity and Security in DevSecOps**
https://blog.gitguardian.com/pipeline-integrity-and-security-in-devsecops/

7 8 18 **Software Supply Chain Attacks To Cost The World $60 Billion By 2025**
https://cybersecurityventures.com/software-supply-chain-attacks-to-cost-the-world-60-billion-by-2025/

9 **2024 State of the Software Supply Chain Report | 10 Year Look Back**
https://www.sonatype.com/state-of-the-software-supply-chain/2024/10-year-look

10 **The biggest supply chain attacks in 2024 | Kaspersky official blog**
https://www.kaspersky.com/blog/supply-chain-attacks-in-2024/52965/

12 **PCI DSS Requirement 11 Explained**
https://pcidssguide.com/pci-dss-requirement-11/

13 **File Integrity Monitoring (FIM), PCI DSS 11.5, and Compliance**
https://atomicorp.com/fim-pci-dss-and-regulatory-compliance/

15 **Set up integrity monitoring - Deep Security Help Center - Trend Micro**
https://help.deepsecurity.trendmicro.com/12_0/aws/set-up-integrity-monitoring.html

16 **What is File Integrity Monitoring? | Best Practices for... - BeyondTrust**
https://www.beyondtrust.com/resources/glossary/file-integrity-monitoring