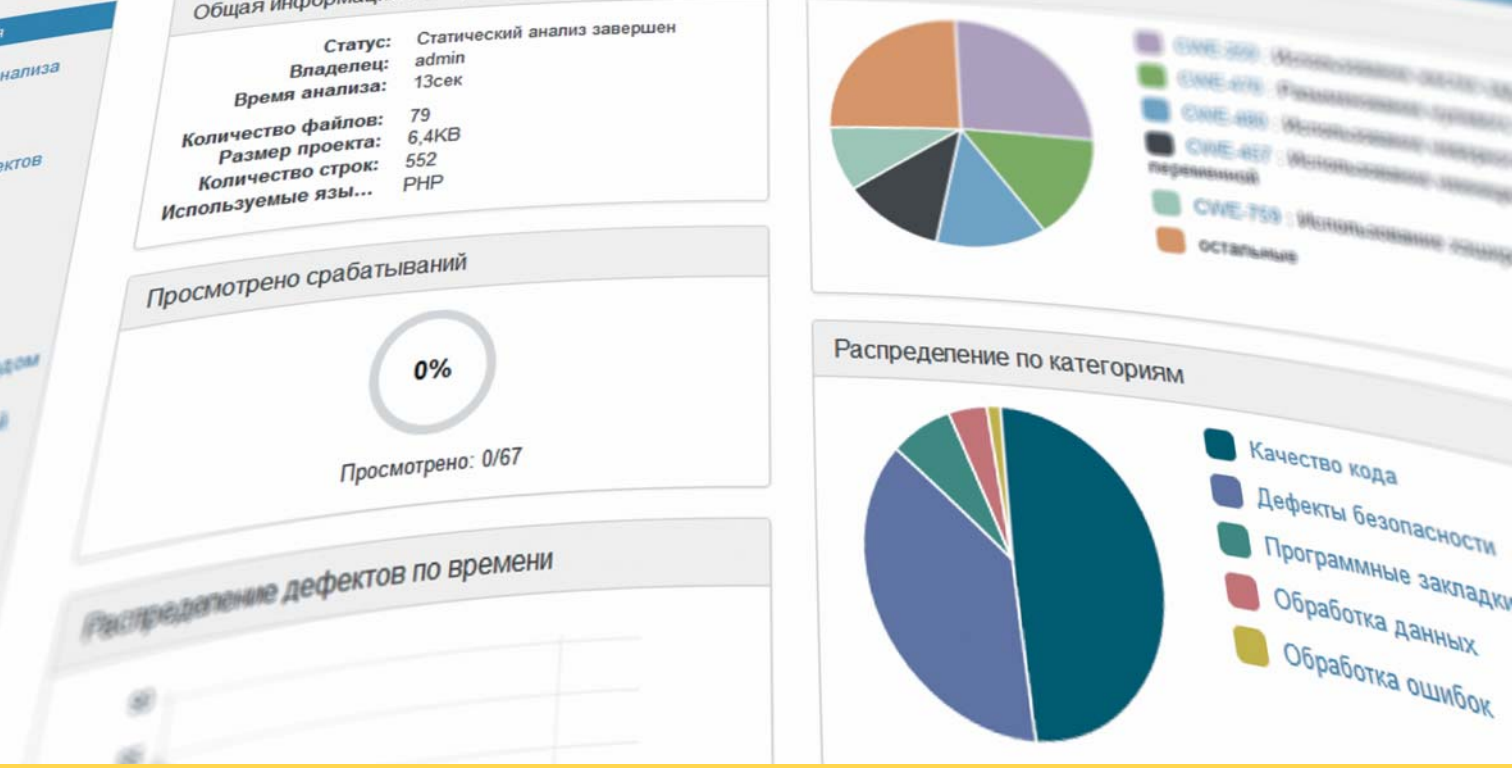

APPCHECKER — ИНСТРУМЕНТ СТАТИЧЕСКОГО АНАЛИЗА

АНДРЕЙ ФАДИН
a.fadin@npo-echelon.ru
СЕРГЕЙ БОРЗЫХ
s.borzykh@npo-echelon.ru
ПАВЕЛ ГУСЕВ
p.gusev@npo-echelon.ru

В статье рассматривается процесс статического анализа безопасности программного кода с целью выявления его дефектов — ошибок и программных закладок. Приведены примеры таких дефектов и описано программное средство для проведения статического анализа.

ОБЛАСТЬ ПРИМЕНЕНИЯ АНАЛИЗАТОРОВ КОДА

В настоящее время информационные технологии проникли практически во все сферы ведения бизнеса. Для решения бизнес-задач создаются крупные, распределенные, постоянно модифицируемые информационные системы с тенденцией к усложнению. Они могут иметь в своем составе как готовые решения, так и внешние IT-сервисы (SaaS), собственные и заказные разработки, бесплатные продукты с открытым исходным кодом (open source). Проблемы в их работе приводят к нарушению информационной безопасности, а как следствие, к финансовым и репутационным потерям бизнеса. По данным исследования [9], последние четыре года финансовые потери бизнеса растут из-за кибератак. Повышение сложности используемых программных комплексов, их включение в контур систем управления государством и производством промышленной продукции требуют постоянного совершенствования методов тестирования, испытаний и контроля программного обеспечения.

Большинство методов анализа ПО, применяемого в аудите безопасности программных систем, можно разделить на две группы (рис. 1): динамические методы (функциональное тестирование) и статические методы (структурное тестирование).

Динамический анализ представляет собой совокупность всех методов анализа программного обеспечения, реализуемых с помощью программ на реальном или виртуальном процессоре. Такие способы наиболее востребованы при исследовании программ методом «черного ящика» (black box), когда имеется доступ лишь к внешним интерфейсам программного обеспечения без учета их структуры, внутренних интерфейсов и состояния [5]. Этот подход не всегда эффективен при поиске ошибок, связанных с комбинациями редко используемых входных данных, а также при выявлении скрытого программного функционала (программных закладок), внесенного туда преднамеренно.

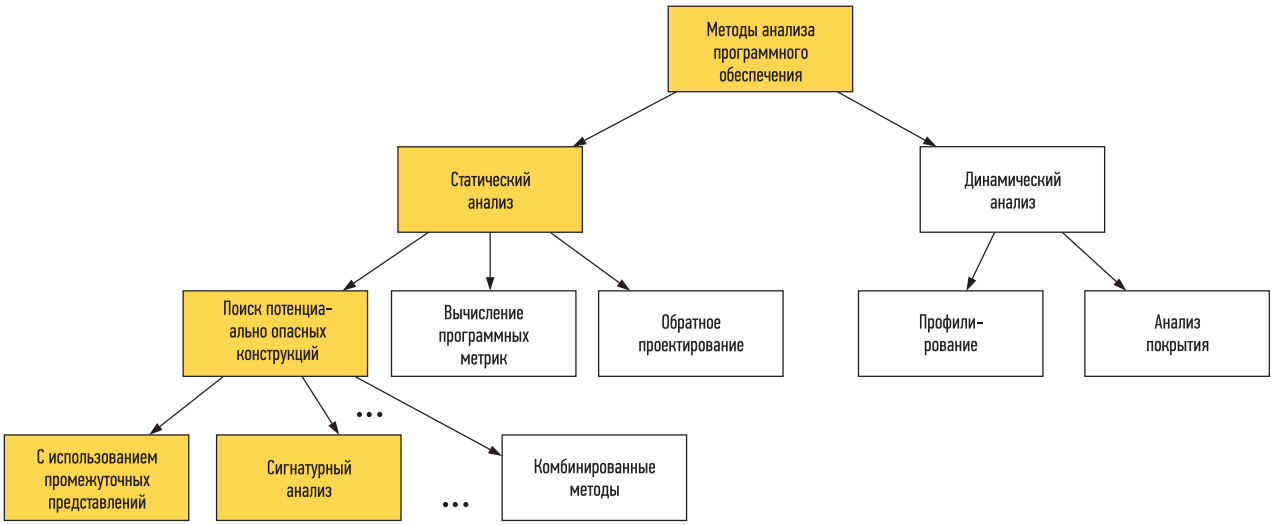

РИС. 1. ▲ Примерная классификация видов анализа программ (цветом выделены те виды анализа, о которых говорится в данной статье)

Для своей работы статический анализ не требует реализации программного кода и допускает полную или частичную автоматизацию процесса, но предполагает доступ не только к загрузочным и объектным модулям, но и к исходным текстам программной системы, к информации, связанной со средой компиляции и выполнением программных компонентов [3–5].

В соответствии с вышеизложенным для одновременного решения задач выявления случайных дефектов кода и программных закладок предлагается использовать методы статического анализа.

МЕТОДЫ СТАТИЧЕСКОГО АНАЛИЗА КОДА

Статический анализ исходных текстов программного обеспечения тесно связан с принципами работы компиляторов. Многие подходы такого анализа основаны на некоторых элементах компиляции, в частности, промежуточное представление исходных текстов в статических анализаторах эволюционирует совместно с развитием теории компиляторов. Современные методы анализа в целях унификации алгоритмов используют различные модели представления кода, например лексический разбор, синтаксическое дерево, дерево Канторовича, графы потока данных и управления и т. д. [1].

Подход, получивший название сигнатурного анализа, подразумевает поиск дефектов в программном коде путем сопоставления фрагментов кода с образцами из базы данных шаблонов (сигнатур) дефектов безопасности. В зависимости от технологии, применяемой при сопоставлении фрагмента кода и шаблона, а также от промежуточного представления, могут использоваться как обычные алгоритмы поиска подстроки в строке, так и регулярные выражения, специальные языки запросов по структурированной информации, в частности XQuery для XML, или специально разработанные для этой задачи способы сопоставления. В [2] приведены примеры правил формирования сигнатур ошибок, соответствующих стандарту CWE. Признакам потенциально опасных конструкций в программных системах и методам оценки их безопасности посвящены статьи 6, 7].

Самый простой способ поиска потенциально опасных конструкций — применение регулярных выражений. Этот вариант достаточно прост в реализации (легко добавляются новые шаблоны уязвимостей), однако имеет несколько критических недостатков. В первую очередь к ним относится невозможность создать шаблоны для сложных дефектов, причем даже для простых дефектов кода регулярное выражение может занимать очень много места, а также очень низкая скорость поиска.

Более эффективным считается построение промежуточного представления кода и его анализ. Именно такой способ используется почти во всех современных анализаторах кода. Общее представление структуры статического анализатора показано на рис. 2.

На начальном этапе решение задачи анализа исходного кода напоминает действия компилятора либо интерпретатора (в случае динамического языка). Исходные тексты разбиваются на лексемы, на основании которых впоследствии строится абстрактное синтаксическое дерево (Abstract syntax tree — AST) и в дальнейшем семантический граф (Abstract semantic graph — ASG). Далее AST

РИС. 2. ▼ Структура статического анализатора

и ASG анализируются механизмом, подобным поиску регулярных выражений в тексте, либо используются скрипты для анализа. Помимо этого, анализ графов выполнения и потоков данных (Data Flow analysis) может дать полезную информацию.

ПРИМЕРЫ ДЕФЕКТОВ КОДА

Самые распространенные виды дефектов кода — ошибки программирования и злонамеренно внесенный код. Следует заметить, что не всякая ошибка программирования приводит к возникновению уязвимости, а только те, в результате эксплуатации которых страдают защищенность, целостность и доступность данных. Злонамеренно внесенный код, как правило, позволяет получить несанкционированный доступ к системе, но бывают и другие типы дефектов, приводящие, например, к удаленному выполнению кода, сбору и отправке конфиденциальной информации либо к реализации несанкционированных действий при наступлении определенного момента времени (time bomb).

К наиболее известным и опасным уязвимостям нужно отнести уязвимости следующих видов:

- отсутствие проверки вводимых данных — может привести к реализации SQL-инъекций, инъекций кода, межсайтового скриптинга, обхода каталогов;
- ошибки работы с памятью — переполнение буфера, необнуленные указатели;
- ошибки реализации системы безопасности — позволяют эксплуатировать давно известные «закрытые» уязвимости.

ОТСУТСТВИЕ ПРОВЕРКИ ВВОДИМЫХ ДАННЫХ

Уязвимость появляется, если не производится проверка данных, поступивших из поля ручного ввода данных; от пользователя ожидается простая информация, например имя, email, текст короткого сообщения. Если при этом в поле ввода дописать код специально сформированной команды SQL или Javascript, то данный код будет выполнен. Например:

```
$name = $_REQUEST['name'];
$res = mysql_query("SELECT *
FROM users WHERE
username ='".$name."'");
```

Если в поле ввода, из которого берется параметр $name, ввести «vasia'; drop table 'users», то таблица users будет удалена, что приведет к потере работоспособности системы. Для предотвращения такого сценария выполнения необходимо использовать безопасный для SQL-инъекции код:

```
$name = $_REQUEST['name'];
$stmt = prepare_query("SELECT *
FROM users WHERE username =?");
$stmt->bind_param($name);
$stmt->execute();
$stmt->bind_result($res);
```

В данном примере каждый параметр передается через отдельный метод bind_param, который и осуществляет фильтрацию специальных символов и ключевых слов.

ОШИБКИ РАБОТЫ С ПАМЯТЬЮ

Чаще всего встречаются ошибки, приводящие к переполнению буфера. Они, как правило, возникают в программах на языках C и C++. Если программа получает данные извне и копирует их во внутренний буфер без проверки размера копируемой области памяти, то при получении достаточно большого объема сведений происходит запись передаваемых данных на место адреса возврата и данных других процедур в области памяти стека. Эта ошибка может быть использована злоумышленником, который подставит нужное значение на место адреса возврата таким образом, чтобы в процессе возврата из подпрограммы переход осуществлялся вовнутрь той же перезаписанной области данных, где можно разместить вредоносный код. Пример уязвимого кода:

```
int main(int argc, char *argv[]) {
    char buf[256];
    strcpy(buf, argv[1]);
    // копирование
    // без проверки длины
}
```

Пример безопасного кода:

```
int main(int argc, char *argv[]) {
    char buf;
    strncpy(buf, argv[1],
sizeof(buf));
// копируется не больше
// размера буфера
}
```

ОШИБКИ РЕАЛИЗАЦИИ

При проектировании и разработке решений для безопасности информационной системы не следует допускать появления в программном коде уязвимостей, которые в дальнейшем могут создать прецеденты нарушения информационной безопасности. К таким уязвимостям относятся:

- хранение паролей в открытом виде, без использования хэш-функций;
- передача данных аутентификации по открытому каналу (HTTP вместо HTTPS);
- отсутствие поддержки CSRF (механизм, предотвращающий XSS-атаки);
- отсутствие проверки качества задаваемых паролей;
- использование устаревших, нестойких криптографических алгоритмов.

ВРЕДОНОСНЫЙ КОД

Один из распространенных примеров вредоносного кода — внедрение разработчиком-злоумышленником возможностей несанкционированного доступа: доступа не только на основании допустимых аутентификационных сведений из базы данных, но и тех данных, о которых знает только разработчик-злоумышленник. Чаще всего к подобной информации относятся пароли-константы, но возможны и пароли, зависящие от даты и времени либо внешних поступающих данных.

Распространены и так называемые логические бомбы, действующие следующим образом: при наступлении определенного момента или при поступлении определенных входных данных программа меняет логику работы на вредоносную.

ЭФФЕКТИВНОСТЬ СТАТИЧЕСКОГО АНАЛИЗА

Сегодня использование статических анализаторов становится объективной потребностью для обеспечения защищенности современных программных систем с высоким уровнем доверия. Введение новых стандартов разработки безопасного ПО подтверждает необходимость эффективных инструментов, поддерживающих качество программного кода.

Использование AppChecker [8] позволяет существенно упростить

и сократить затраты на ревизию кода при разработке программ, а также повысить качество кода и сократить количество ошибок и уязвимостей еще на этапе проектирования, что также позитивно скажется на затратах на доработку и поддержку выпускаемого ПО.

Помимо основного функционала в любом программном обеспечении, взаимодействующем с пользователем, очень важно удобство его применения. Удобный и понятный интерфейс (рис. 3), интеграция в процесс разработки, простота установки, обновления и поддержки становятся важными характеристиками анализаторов кода. Тем не менее сделать анализатор полностью автоматическим, действующим по «нажатию одной кнопки», невозможно.

Как уже упоминалось, работа анализатора во многом близка работе компилятора. И для того чтобы провести качественный анализ, требуются те же действия, которые выполняет система сборки: проанализировать зависимости в проекте, распознать и «подключить» библиотечные функции и т. д., что может потребовать дополнительных настроек процесса анализа. В сравнении с другими статическими анализаторами, AppChecker предоставляет пользователям бóльшую гибкость в настройке: он понятен простым пользователям, а опытные разработчики смогут интегрировать его в процесс сборки любой сложности.

Важной составляющей безопасности кода является применение проверенных сторонних компонентов — библиотек или готовых программ с открытым исходным кодом. Их можно проверить, сообщая авторам о найденных уязвимостях и создавая базу уязвимостей open-source библиотек. Это особенно актуально для поддерживаемых анализатором AppChecker интерпретируемых языков PHP, Javascript, в которых программы хранятся в виде исходных текстов.

ЗАКЛЮЧЕНИЕ

В статье рассмотрен метод статического анализа исходных кодов программного обеспечения, а также основные задачи и проблемы реализации автоматизированной проверки кода на примере статического анализатора AppChecker. Авторы данного продукта разработали эту технологию для удовлетворения потребностей рынка средств управления качеством исходного кода программ. ⬢

ЛИТЕРАТУРА

1. Компиляторы: принципы, технологии и инструментарий / Ахо А. В. [и др.]. М.: ООО «И. Д. Вильямс». 2016. 1184 с.
2. Марков А. С., Фадин А. А. Систематика уязвимостей и дефектов безопасности программных ресурсов // Защита информации. Инсайд. 2013. № 3 (51). С. 56-61.
3. Markov A., Fadin A., Shvets V., Tsirlov V. The experience of comparison of static security code analyzers, International Journal of Advanced Studies. 2015. Т. 5. № 3. С. 55-63.
4. Markov A.S., Fadin A.A., Tsirlov V.L. Multilevel Metamodel for Heuristic Search of Vulnerabilities in the Software Source Code, International Journal of Control Theory and Applications. 2016. Т. 9. № 30. С. 313-320.
5. Аветисян А. И., Белеванцев А. А., Чукляев И. И. Технологии статического и динамического анализа уязвимостей программного обеспечения // Вопросы кибербезопасности. 2014. № 3 (4). С. 20-28.
6. Жидков И. В., Кадушкин И. В. О признаках потенциально опасных событий в информационных системах // Вопросы кибербезопасности. 2014. № 1 (2). С. 40-48.
7. Рибер Г., Малмквист К., Щербаков А. Многоуровневый подход к оценке безопасности программных средств // Вопросы кибербезопасности. 2014. № 1 (2). С. 36-39.
8. Релиз версии 1.5.10 / ООО «Эшелон Инновации». 2015. www.iechelon.ru/version_1-5-10/.
9. 2016 Cyber Crime Study & the Risk of Business Innovation. Ponemon Institute. www.ponemon.org.

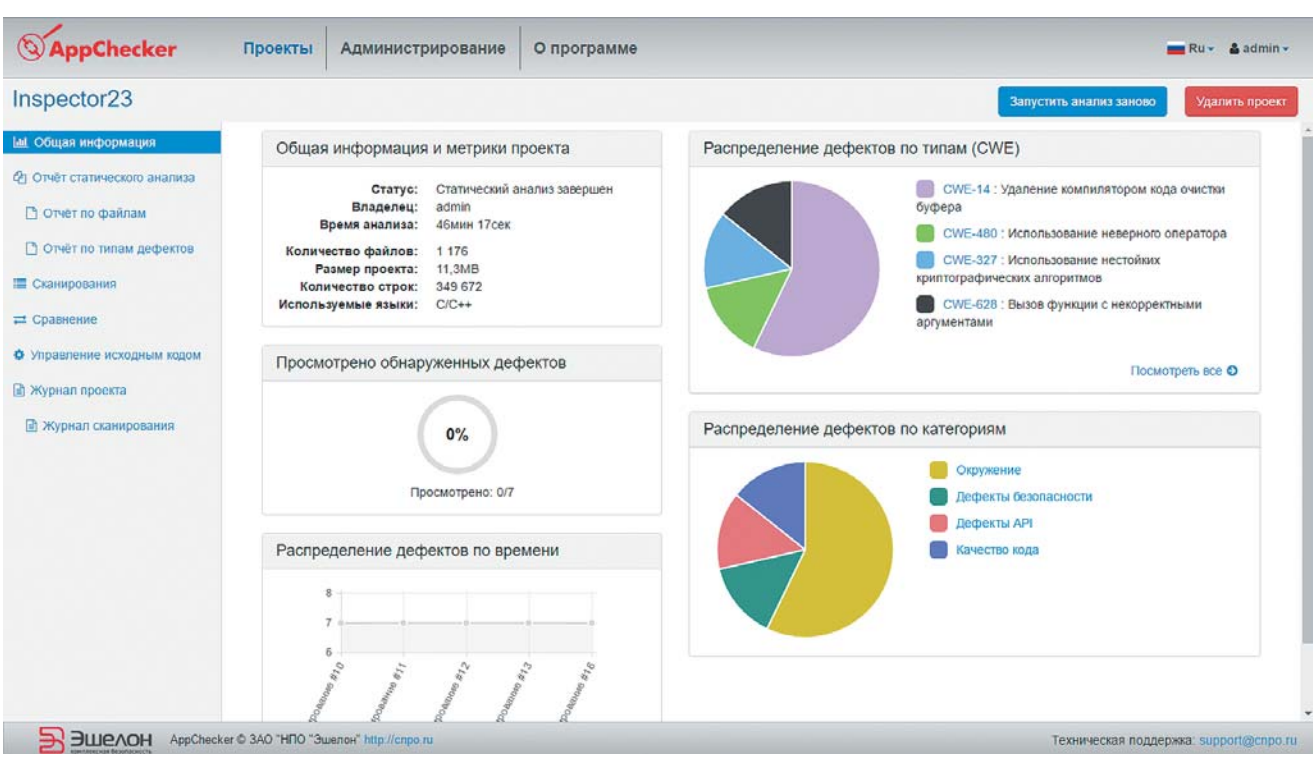

РИС. 3. ▼
Общий вид рабочего экрана AppChecker