Программные уязвимости – на стыке аппаратуры и программного обеспечения

Лекция 4. Поиск уязвимостей. Разработка безопасного кода.*

*Использованы материалы Tal Garfinkel (Stanford), Jonathan Howell (MSU), Microsoft SDL Resources

Обнаружение уязвимостей

- На открытом рынке неопубликованная уязвимость может принести 500$-100,000$ и более
- Хорошие специалисты по поиску уязвимостей зарабатывают 180-250/час за консультации
- Немногие компании могут найти хороших сотрудников, многие даже не знают, что это возможно
- Во многом всё ещё «чёрная магия»

Почему нельзя устранить все уязвимости сразу?

- Нереализуемо на практике
  - Формальная верификация в общем случае сложна, для больших задач иногда невозможна.
- Почему не писать программы на Java, Haskell, <ваш любимый безопасный язык>
  - Не решает всех проблем
  - Производительность, существующий code base, гибкость, профессионализм программистов, и т.д.
- Не выгодно
  - По-настоящему заинтересованы в поиске уязвимостей только «плохие парни»
- Инкрементальные решения стимулируют использование инкрементальных решений
  - Появление средств обнаружения ошибок делают экономиечски невыгодными радикальные решения, построенные «с нуля»

Особенности ошибок

- Программисты повторяют ошибки
  - Copy/paste
  - Недопонимание API
    - т.е. драйверы linux, небезопасные функции работы со строками
  - Отдельные программисты повторяют собственные ошибки
- Ошибки появляются из-за неверных или изменившихся допущений
  - Доверенный ввод становится недоверенным
- Чужие ошибки становятся вашими
  - Open source, код сторонних приложений

Арсенал поиска ошибок

- Моделирование угроз: изучить архитектуру, выписать возможные пути, как сделать что-то не так.
- Ручной аудит кода
  - Ревью кода
- Автоматизированные средства
- Все эти методики дополняют друг друга
  - Очень мало простых решений, ни одного универсального

Моделирование архитектурных угроз: базовый подход к анализу рисков

- Цель – обнаружение уязвимостей проектирования
- Анализ начинается с построения высокоуровневой модели системы
- Анализируем:
  - Модель нарушителя – кто и зачем может атаковать нашу систему
  - Риски на каждом уровне архитектуры
  - Классы уязвимостей, которые могут присутствовать в каждом из компонентов, зависимости по данным
  - Оценка ущерба в случае успешной реализации угрозы
  - Вероятность реализации угрозы
  - Список возможных мер противодействия успешной реализации угрозы

Пример комментария в коде

```
//   LCG - Linear Conguential Generator
//   used to generate ballot serial numbers
//   A psuedo-random-sequence generator
//   (per Applied Cryptography,
//   by Bruce Schneier, Wiley, 1996)
```

Unfortunately, linear congruential generators cannot be used for cryptography”

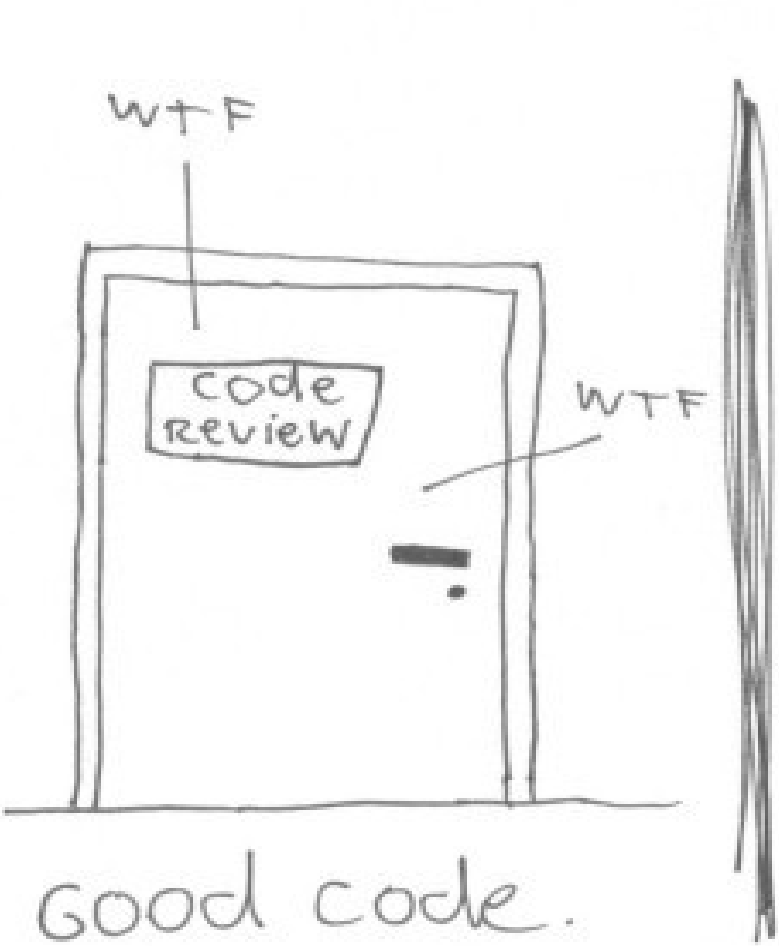
Page 369

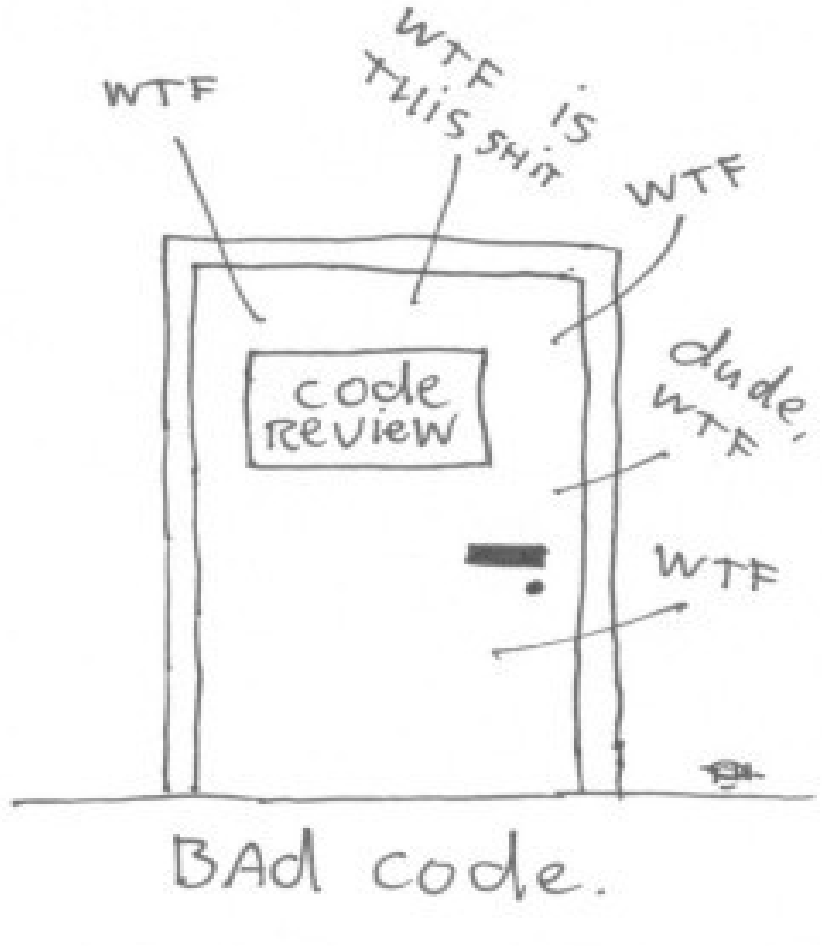
Applied Cryptography, by Bruce Schneier

- BallotResults.cpp
Diebold Election Systems

Тестирование и ревью кода

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE





Компонентное (модульное, unit-) тестирование

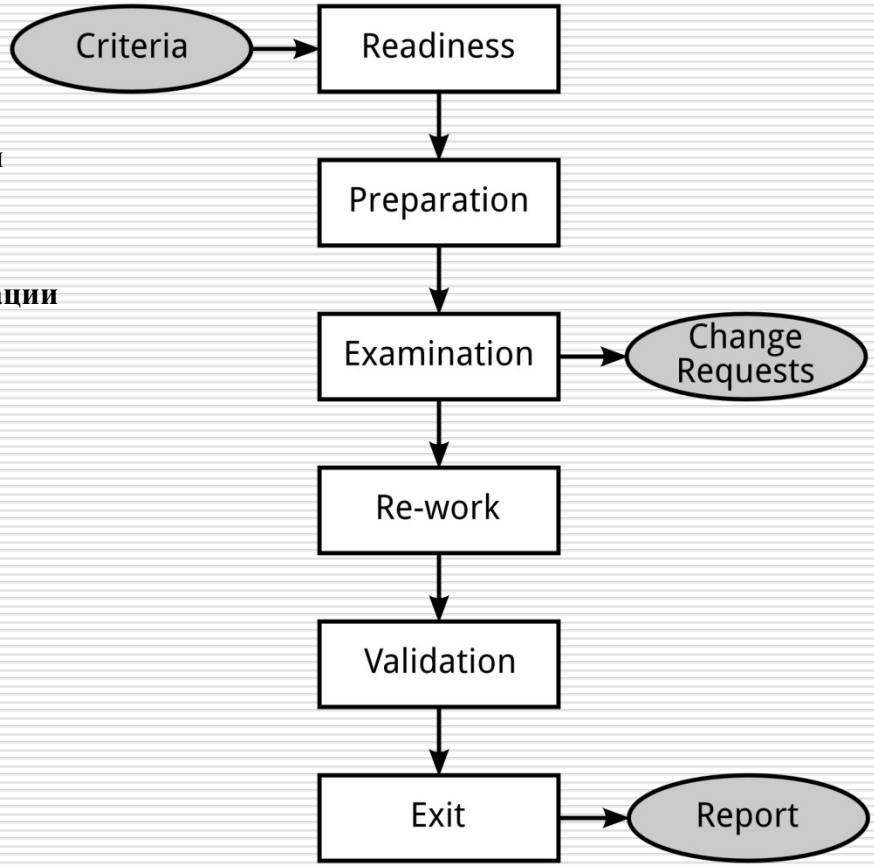
- Статическое юнит-тестирование
  - Код анализируется для всех возможных путей исполнения
  - Код каждого компонента проверяется на соотвествие требованиям к данному компоненту
- Динамическое юнит-тестирование
  - Компонент выполняется и результаты выполнения анализируются
  - На основе анализа поведения системы делаются выводы о качестве кода
- Статическое тестирование не заменяет динамического
- Рекомендуется статическое тестирование выполнять до динамического

Статическое тестирование

- При статическом тестировании кода выполняются:
  - **Инспектирование:** пошаговый групповой анализ рабочей версии продукта, на каждом шаге анализируется соответствие некоторым предопределённым критериям
  - **Прогоны:** автор демонстрирует команде ревьюеров работу продукта на наборе заранее определённых и документированных сценариев
- Основная идея заключается в систематическом анализе исходного кода
- **Цель анализа кода – оценить качество кода, а не его автора**
- Анализ кода должен быть запланирован
- Ключ к успешному ревью кода – принцип «разделяй и властвуй»
  - Ревьюер анализирует небольшую часть кода компонента в изолированном окружении
    - В таком случае он ничего не упустит
    - Из корректности всех составляющих следует корректность всего компонента

Статическое компонентное тестирование (ревью кода)

- Шаг 1: **Начало**
  - **Критерии**
    - **Полнота**
    - **Наличие минимума функциональности**
    - **Читабельность**
    - **Сложность**
    - **Наличие требований к ПО и документации**
  - **Роли**
    - **Модератор**
    - **Автор**
    - **Ведущий**
    - **Хранитель записей**
    - **Ревьюеры**
    - **Наблюдатель**
- Шаг 2: **Подготовка**
  - **Список вопросов**
  - **Потенциальный Change Request (CR)**
  - **Предполагаемые возможности улучшения**



Шаги ревью кода

Статическое компонентное тестирование (ревью кода)

- Шаг 3: **Анализ**
  - Автор представляет код
  - Ведущий читает код
  - Хранитель записей документирует CR
  - Модератор следит за тем, что ревью выполняется
- Шаг 4: **Обработка**
  - Подготовка списка всех CR
  - Подготовка списка всех улучшений
  - Протокол обсуждения
  - Автор работает над всеми CR для исправления проблемы
- Шаг 5: **Валидация**
  - Все CR проходят независимую проверку
- Шаг 6: **Окончание**
  - Рассылка сводного отчёта по результатам обсуждения

Change Request (CR) включает:

- Краткое описание проблемы
- Приоритет **CR**
- Имя ответственного
- Дедлайн для рассмотрения **CR**

Статическое компонентное тестирование (ревью кода)

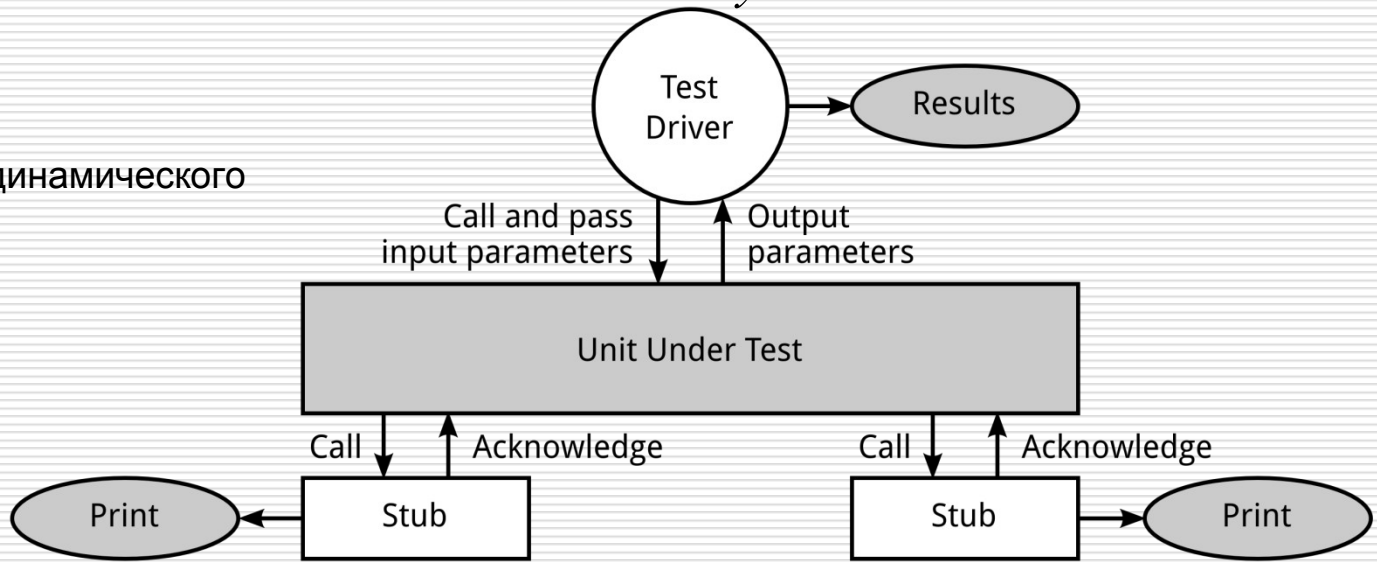
По результатам ревью кода должны быть собраны метрики:

- Количество проанализированных строк кода (LOC) в час
- Количество CR на тысячу строк кода (KLOC)
- Количество CR в час
- Общее время, затраченное на ревью

Динамическое компонентное тестирование

- Окружение компонента эмулируется, запуск в изолированном окружении
- Вызывающий компонент называется *test driver*
  - *test driver* это программа которая запускает компонент в тестовом окружении
  - Она подаёт данные на вход компоненту и записывает результат
- Эмулируемые компоненты называются *заглушками*

Окружение для динамического тестирования



Тестирование на основе мутаций

- Модифицировать программу внесением одного простого изменения
- Модифицированная программа называется *мутантом*
- Мутант считается *убитым,* когда завершается ошибкой на некотором тесте.
- Мутант считается *эквивалентным* исходной программе, если на одних и тех же входных данных он генерирует идентичный исходной программе выход
- Счётчик мутаций для набора тестов – доля неэквивалентных мутантов, убитых тестовым набором
- Тест называется *устойчивым к мутациям* если его счётчик мутаций равен 100%

Тестирование на основе мутаций

Пусть у нас есть программа P:

```
1.  main(argc,argv)
2.  int argc, r, i;
3.  char *argv[];
4.  { r = 1;
5.  for (i = 2 ; i<=3; i++) {
6.  if (atoi(argv[i]) > atoi(argv[r])) r = i;
7.  printf("Value of the rank is %d \n", r);}
8.  exit(0); }
```

Набор 1: вход: 1 2 3
выход: Value of the rank is 3

Набор 2: вход: 1 2 1
выход: Values of the rank is 2

Набор 3: вход: 3 1 2
выход: Value of the rank is 1

Мутант 1: Изменить строку 5 на for (i = 1 ; i<=3; i++) {
Мутант 2: Изменить строку 6 на if (i > atoi(argv[r])) r = i;
Мутант 3: Изменить строку 6 на if (atoi(argv[i]) >= atoi(argv[r])) r = i;
Мутант 4: Изменить строку 6 на if (atoi(argv[r]) > atoi(argv[r])) r = i;
Выполнив мутантов на тестовом наборе, получим:
Мутанты 1 и 3 переживут тестовый набор, т.е. они устойчивы к нему.
Мутант 2 будет убит набором 2.
Мутант 1 будет убит наборами 1 и 2
Счётчик мутаций 50%, предполагая что мутанты1 и 3 не эквиваленты

Тестирование на основе мутаций

- Счётчик мал, т.к. мы предполагаем мутантов1 и 3 не эквивалентными
- Надо показать, что мутанты 1 и 3 эквивалентны или убиваемы
- Чтобы сделать их убиваемыми, нужно добавить тестовый пример
- Сначала проанализируем мутант 1 чтобы получить тест. Разница между мутантом и исходной программой в стартовой точке.
- Мутант 1 начинает с i = 1, тогда как P начинает с i = 2. На результат это не влияет. Поэтому мы делаем заключение, что мутант 1 эквивалентен P.
- Теперь добавим четвёртый тестовый набор:

Набор 4:

вход: 2 2 1

- Программа P сгенерирует выход “Value of the rank is 1” и мутант 3 сгенерирует выход “Value of the rank is 2”
- Теперь тестовый набор убивает мутанта 3, что даёт нам счётчик 100%

Тестирование на основе мутаций

Тестирование на основе мутаций делает два важных предположения:

- Гипотеза компетентности программиста
  - Программисты, как правило, компетентны, и поэтому не создают *случайных* программ
- Эффекты сопряжения
  - Сложные ошибки сопрягаются с простыми ошибками таким образом, что тестовый набор, обнаруживающий простые ошибки, сможет обнаружить большинство сложных ошибок

Статический анализ

Два типа статического анализа

- … который вы пишете в 100 строк на python.
  - Ищем небезопасные функции работы со строками- `strncpy(), sprintf(), gets()`
  - Ищем небезопасные функции из вашего собственного кода
  - Ищем повторение проблемного кода (небезопасные интерфейсы, copy/paste плохого кода, и т.д.)
- … на котором вы защищаете кандидатскую
  - Покупаем у coverity, fortify, и т.д.
  - Используем встроенный в visual studio
  - Реализуем собственный поверх LLVM

Основы статического анализа

- Придумываем абстрактное модельное представление свойств программы, ищем возможные проблемы
- Средства из области анализа программ
  - Приведение типов, анализ потока данных, доказательство теорем
- Обычно на основе исходного кода, также может быть байткод или результат дизассемблирования
- Достоинства
  - Полное покрытие (в теории)
  - Потенциально можно проверить отсутствие или найти все экземпляры ошибок заданного класса
- Недостатки
  - Ложные срабатывания
  - Многие свойства нелегко промоделировать
  - Сложно реализовать
  - В реальных системах почти никогда нет полного исходного кода (операционная система, разделяемые библиотеки, динамически порождаемый код, и т.д.)

Пример: где ошибка?

```
int read_packet(int fd)
{
    char header[50];
    char body[100];
    size_t bound_a = 50;
    size_t bound_b = 100;

    read(fd, header, bound_b);
    read(fd, body, bound_b);

    return 0;
 }
```

Пример: где ошибка?

```
int read_packet(int fd)
{
    char header[50]; //модель (header, 50)
    char body[100];  //модель (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

    read(fd, header, 100);
    read(fd, body, 100);

    return 0;
 }
```

Пример: где ошибка?

```
int read_packet(int fd)
{
    char header[50]; //модель (header, 50)
    char body[100];  //модель (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

    read(fd, header, 100); //распространение константы
    read(fd, body, 100);   //распространение константы

    return 0;
 }
```

Пример: где ошибка?

```
int read_packet(int fd)
{
    char header[50]; //модель (header, 50)
    char body[100];  //модель (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

   //проверяем условие read(fd, dest.size >= len)
    read(fd, header, 100); //распространение константы
    read(fd, body, 100);   //распространение константы

    return 0;
 }
```

Пример: где ошибка?

```
int read_packet(int fd)
{
    char header[50]; //модель (header, 50)
    char body[100];  //модель (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

   //проверяем условие read(fd, 50 >= 100) => несовпадение
   размера!!
    read(fd, header, 100); //распространение константы
    read(fd, body, 100);   //распространение константы

    return 0;
 }
```

Редко бывает настолько просто

- Нужна информация о связях между функциями
- Неоднозначность из-за использования указателей
- Недостаточная связь между размером и типом
- Недостаточно информации о входе программы и её состоянии…

Статический анализ не панацея, но очень полезен при правильном использовании

Правильный уход за средствами статического анализа

- Поиск исправление ошибок рано и часто
  - иначе количество ложных срабатываний будет мешать поиску ошибок
- Использование аннотаций
  - Поможет найти больше ошибок с меньшим числом ложных срабатываний (SAL)
- Написание собственных правил
- Использование возможностей компилятора
  - `gcc -Wall, /analyze in visual studio`
- Встраивание этого в систему сборки

Простейшие средства статического анализа

- Splint http://splint.org/
- Sparse http://kernel.org/pub/linux/kernel/people/
- Cppcheck http://sourceforge.net/apps/mediawiki/cpp
- Flawfinder http://www.dwheeler.com/flawfinder

Пример использования

```
1.  /*basicheap.c*/
2.  int
3.  main(int argc, char** argv) {
4.          char *buf;
5.          char *buf2;
6.    buf=(char*)malloc(1024);
7.    buf2=(char*)malloc(1024);
8.    printf("buf=%p buf2=
      %p\n",buf,buf2);
9.    strcpy(buf,argv[1]);
10.   free(buf2);
11. }
```

```
Flawfinder version 1.27, (C) 2001-2004 David A.
Wheeler.
Number of dangerous functions in C/C++
ruleset: 160
Examining basicheap.c
basicheap.c:9:  [4] (buffer) strcpy:
  Does not check for buffer overflows when
copying to destination.
  Consider using strncpy or strlcpy (warning,
strncpy is easily misused).

Hits = 1
Lines analyzed = 11 in 0.51 seconds (962
lines/second)
```

Динамический анализ

Стандартный динамический анализ

- Запуск программы в инструментированном окружении
  - Бинарный транслятор, статическое инструментирование кода, эмулятор
- Поиск ошибок
  - Использование недоступной памяти, гонки, разыменование нулевого указателя, и т.д.
- Примеры: Purify, Valgrind, обычные проверки в операционной системе (проверки libc)

Регрессия vs. фаззинг

- Регрессионное тестирование: запуск программы на большом количестве примеров нормального входа.
  - Цель: минимизировать вероятность отказа для легитимного пользователя (т.е. проверки на плохой ввод)
- Фаззинг: запуск программы на большом числе некорретных входных данных, поиск ошибок.
  - Цель: предотвратить обнаружение злоумышленником уязвимостей, которые можно эксплуатировать

Основы фаззинга

- Автоматическая генерация тестовых примеров
- Множество слегка аномальных примеров подаются на вход программе
- Наблюдаем за появлением ошибок
- Вход обычно представлен файлами (.pdf, .png, .wav, .mpg)
- Или сетевым взаимодействием…
  - HTTP, SNMP, SOAP

Простой пример

- Standard HTTP GET request
  - GET /index.html HTTP/1.1
- Anomalous requests
  - AAAAAA...AAAA /index.html HTTP/1.1
  - GET ///////index.html HTTP/1.1
  - GET %n%n%n%n%n%n.html HTTP/1.1
  - GET /AAAAAAAAAAAAA.html HTTP/1.1
  - GET /index.html HTTTTTTTTTTTTP/1.1
  - GET /index.html HTTP/1.1.1.1.1.1.1.1

Способы генерации тестовых примеров

- На основе мутации - “Тупой фаззинг”
- На основе генерации - “Умный фаззинг”

Фаззинг на основе мутации

- Не используется почти никакой информации о структуре входных данных
- Аномалии добавляются в заданные нормальные примеры данных
- Аномалии могут быть совершенно случайными или использовать простую эвристику (удаление NUL, сдвиг символа вперёд)
- Примеры средств:
  - Taof, GPF, ProxyFuzz, FileFuzz, Filep,…

Пример: фаззинг программы просмотра pdf

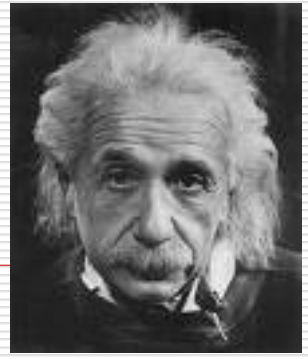
- Поискать в Yandex файлы с расширением .pdf (миллиард страниц)
- Обойти сколько-то страниц и набрать базу файлов
- Использовать фаззер (или написать скрипт), чтобы
  1. Взять файл
  2. Изменить его слегка
  3. Подать на вход программе
  4. Если программа упала, то запомнить на чём.

«Тупой» фаззинг кратко

- Достоинства
  - Очень легко реализовать и автоматизировать
  - Почти не требуется знание протокола
- Недостатки
  - Ограниченные возможности в силу самого подхода
  - Не сработает в случае протоколов с обнаружением и коррекцией ошибок, протоколов со схемой запрос-ответ, и т.д.

Фаззинг на основе генерации примеров

- Тестовые примеры генерируются на основе какого-то представления формата: RFC, документации, и т.д.
- В каждое допустимое место добавляются аномалии.
- Знание протокола должно дать больше возможностей для обнаружения ошибок.



Пример: описание протокола

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);     // Width
                s_push_int(0x14, 1);     // Height
                s_push_int(0x8, 3);      // Bit Depth - should be 1,2,4,8,16, based on
    colortype
                s_push_int(0x3, 3);      // ColorType - should be 0,2,3,4,6
                s_binary("00 00");       // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);      // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Фаззинг на основе генерации примеров вкратце

- Достоинства
  - Полнота
  - Могут работать со сложными зависимостями, например с контрольными суммами
- Недостатки
  - Нужно иметь спецификацию протокола
    - Часто можно найти хорошие средства для существующих протоколов, т.е. HTTP, SNMP
  - Для сложных протоколов написание фаззера очень трудоёмко
  - Спецификация не эквивалентна коду

Средства для фаззинга

Генерация входа

- Существующие генерирующие фаззеры для распространённых протоколов (FTP, HTTP, SNMP, и т.д.)
  - Mu-4000, Codenomicon, PROTOS, FTPFuzz
- Фреймворки: вы пишете спецификацию, они генерируют тестовый набор
  - SPIKE, Peach, Sulley
- Автоматизация «тупого» фаззинга: вы задаёте пакеты или файлы, они генерируют тестовые наборы
  - Filep, Taof, GPF, ProxyFuzz, PeachShark
- Также существует много специальных фаззеров
  - ActiveX (AxMan), регулярные выражения, и т.д.

Как подать набор на вход программе

- Самый простой вариант
  - Запуск программы на файле из набора
  - Проигрывание дампа из набора
- Модификация программы/клиента
  - Вызвать фаззер в нужной точке
- Использовать фреймворк
  - Например, Peach автоматизирует создание фаззеров для COM-интерфейса

Обнаружение ошибки

- Смотрим, завершилась ли программа некорректно
  - Тип ошибки может сказать о многом (SEGV vs. assert fail)
- Запустить программу под отладчиком для динамической памяти (valgrind/purify)
  - Обнаруживает больше ошибок, но дороже в запуске.
- Смотрим, зависла ли программа
- Запускаем собственные анализаторы, например плагины к valgrind

Автоматизация фаззинга

- Некоторые средства - Sulley, Peach, Mu-4000 – позволяют автоматизировать запуск, запись поведения программы, и т.д.
- Виртуальные машины помогают создать воспроизводимые условия тестирования
- Некоторое количество ручной работы тем не менее требуется

Сколько фаззинга достаточно?

- Фаззеры на основе мутации могут генерировать бесконечное количество примеров... Когда остановиться?
- Фаззеры на основе генерации создают конечный набор примеров. Что если они все отработали, но ошибок не найдено?

Пример: PDF

- У нас есть файл PDF размером 248,000 байтов
- Есть один байт, изменение значения которого может иногда вызвать ошибку
  - Этот байт где-то в районе 94% длины файла
- Одна случайная мутация файла даёт вероятность ошибки .00000392
- В среднем, требуется 127,512 тестовых примеров, чтобы её найти
- Если один тест занимает 2 секунды, то это всего 3 дня...
- А может занять неделю или больше...

Покрытие кода

- Некоторые ответы на эти вопросы лежат в области *покрытия кода*
- Покрытие кода это метрика, которая позволяет определить, какой объём кода был исполнен.
- Данные об этом могут быть собраны с помощью различных средств профилировки, например gcov

Типы покрытия

- Строковое покрытие
  - Измеряет, какое количество строк было исполнено.
- Покрытие веток
  - Измеряет, какое количество веток кода было затронуто (условные переходы)
- Покрытие путей
  - Измеряет, какое количество путей было затронуто.

Пример

```
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```

- Требует
  - 1 тест для покрытия строк
  - 2 теста для покрытия веток
  - 4 теста для покрытия путей
    - т.е. `(a,b) = {(0,0), (3,0), (0,3), (3,3)}`

Проблемы обеспечения полноты покрытия

- Можно добиться покрытия, но не найти ошибку

```
mySafeCpy(char *dst, char* src){
    if(dst && src)
        strcpy(dst, src);
}
```

- Проверка ошибок обычно пропускается (и нас это особо не волнует)

```
ptr = malloc(sizeof(blah));
if(!ptr)
    ran_out_of_memory();
```

- Достижима только «атакуемая поверхность» (“attack surface”)
  - т.е. код, обрабатывающий контролируемый пользователем ввод
  - Простого способа измерить поверхность нет
    - Возможно, использовать статический анализ?

Правила хорошего фаззинга

- Знание особенностей протокола очень помогает
  - Генерация по спецификации лучше случайной, более точная спецификации ведёт к лучшему фаззингу
- Чем больше фаззеров, тем лучше
  - Реализации отличаются друг от друга, типы ошибок, которые они находят, также различаются
- Чем дольше работает фаззер, тем больше ошибок находит
- Лучшие результаты получаются при направленном процессе
  - Если застряли, переключайтесь на ручную профилировку
- Анализ полноты покрытия очень помогает в направленном фаззинге

Перспективные методы

Нерешённые проблемы

- Что если у нас нет спецификации или описания протокола, как избежать создания спецификации с нуля?
- Как более точно выбрать, какие тестовые примеры генерировать

Фаззинг методом «прозрачного ящика»

- Выявить спецификацию протокола по прогонам, после чего применить фаззинг на основе генерации примеров

Как генерировать ограничения?

- Наблюдаем за программой
  - Инструментированный код (EXE)
  - Бинарные трансляторы (SAGE, Catchconv)
- Интерпретируем вход как поток символов
- Выводим ограничения

Пример:

```
int test(x)
{
    if (x < 10) {
                    //X < 10 and X <= 0 gets us this path
     if (x > 0) {
                    //0 < X < 10 gets us this path
          return 1;
    }
  }
                //X >= 10 gets us this path
  return 0;
}
```

Ограничения:

X >= 10

0 < X < 10

X <= 0

Разрешая ограничения, получаем тестовые примеры: {12,0,4}

- Обеспечивает максимальное покрытие

Анализ «серого ящика»

- Генетический фаззинг
- Направленные мутации на основе метрики приспособленности
- Предпочитаем те мутации, которые обеспечивают
  - Лучшее покрытие
  - Модифицируют вход для потенциально опасных функций (например, memcpy)
- EFS, autodafe

Microsoft SDL

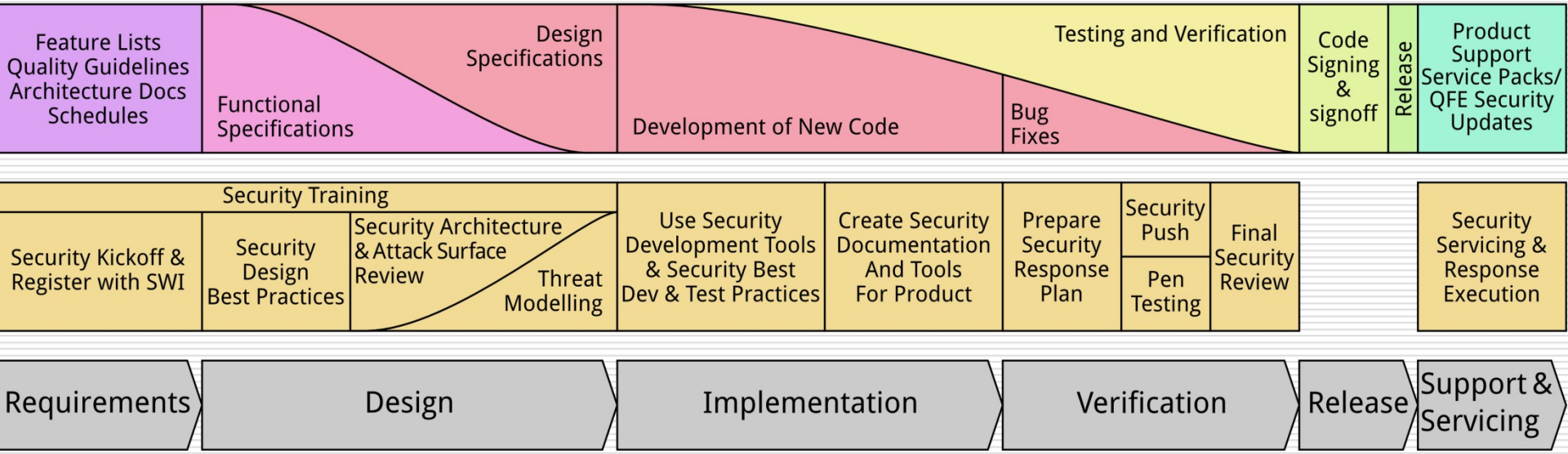
Microsoft Security Development Lifecycle

- “The Trustworthy Computing Security Development Lifecycle”
  http://msdn.microsoft.com/en-us/library/ms995349
  - Авторы:
    - Steve Lipner
    - Michael Howard

$SD^3 + C$

- Secure by Design
- Secure by Default
- Secure in Deployment
- Communications

Процессы SDL



Основной процесс SDL

Фазы SDL

- Фаза разработки требований
- Фаза проектирования
- Фаза реализации
- Фаза верификаци
- Фаза релиза
- Фаза поддержки

Фаза разработки требований

- Консультант по безопасности
  - Основные вехи
  - Критерий завершения
- Документация
- Функциональные требования к безопасности

Фаза проектирования

- Определение основных рекомендаций по обеспечению безопасности архитектуры ПО
- Документирование всех потенциальных точек «атакуемой поверхности» ПО
- Моделирование угроз
- Определить дополнительные критерии безопасности при релизе (обновлении) ПО

Фаза реализации

- Применение стандартов безопасного программирования и тестирования
- Применение средств тестирования безопасности, включая фаззеры
- Применение статических анализаторов
- Ревью кода

Фаза верификации

- “Security push”:
  - Ревью безопасности
  - Тестирование наиболее приоритетного кода

Фаза релиза

- Итоговый ревью безопасности
  - Независимый (сторонний) ревью
  - Выполняется главной командой по безопасности
  - Пентест
  - Результат не бинарный (пройден/не пройден) – даётся итоговая оценка качества продукта перед релизом

Фаза поддержки

- Анализ сообщений от пользователей
- Выпуск адвайзори
- Выпуск обновлений
- Обновление средств статического и динамического анализа кода

SDL в Microsoft

- Обязательность применения
- Обязательность повышения квалификации в области безопасности
- Метрики для команды разработчиков
- Отдельная центральная команда по безопасности

Обязательное применение

- Применение SDL обязательно для ПО:
  - Обрабатывающего персональные данные пользователей
  - Используемого в корпоративном сегменте
  - Имеющего выход в сеть или Интернет
- Безопасность платформы/операционной системы

Обязательное повышение квалификации

- Область информационной безопасности часто меняется
- Ежегодное обучение информационной безопасности
- Книги:
  - Моделирование угроз
  - Безопасное проектирование

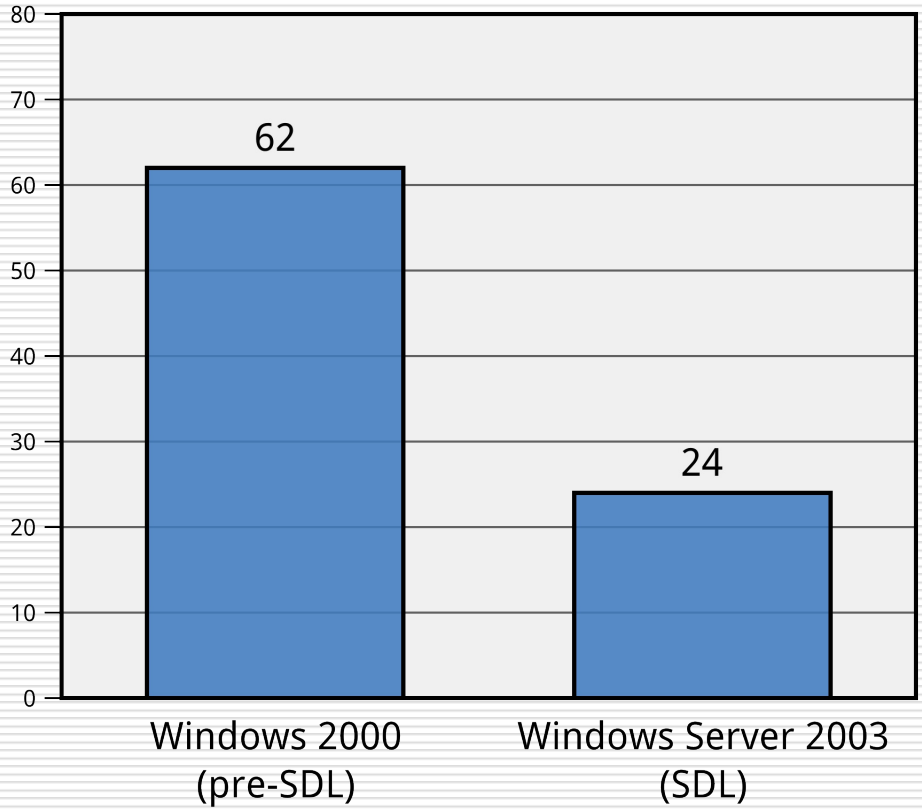
Метрики

- “невозможно управлять тем, что нельзя измерить”
- Метрики для разработчиков
  - Покрытие персонала тренингами
  - …
  - Количество уязвостей в ПО, найденных после релиза
- Агрегированные метрики для менеджмента

Центральная команда по безопасности

- Secure Windows Initiative (SWI)
  - Разработка, поддержка, улучшение SDL
  - Разработка и улучшение курсов повышения квалификации
  - Предоставление “консультантов по безопасности”
  - Эксперты в конкретных прикладных областях безопасности
  - Выполняют итоговый анализ защищённости

Первые результаты в Microsoft



Наблюдения по результатам первых лет использования

- Моделирование угроз – важный этап
- Ревью кода, автоматизированные средства анализа кода, фаззинг
- Пентесты
- Инвестиции в безопасность

Вопросы?