Очевидно, что, следуя описанной методике, аналогичным образом можно получить оценки нормы ошибок аутентификации EER и оптимальные значения коэффициентов *t* для любых других методов классификации, параметров областей распределения и мерности пространства входных данных.

Работа поддержана грантами РФФИ: № 06-07-89010; № 06-07-96609

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Иванов А.И.* Биометрическая идентификация личности по динамике подсознательных движений. – Пенза: Изд-во Пенз. гос. ун-та, 2000. – 188 с.

2. *Брюхомицкий Ю.А., Казарин М.Н.* Метод биометрической идентификации пользователя по клавиатурному почерку на основе разложения Хаара и меры близости Хэмминга / Известия ТРТУ. Тематический выпуск «Материалы V Международной научно-практической конференции «Информационная безопасность».-Таганрог: Изд-во ТРТУ, 2003. № 4(33). – С. 141-149.

3. *Поллард Дж.* Справочник по вычислительным методам статистики / Пер. с англ. В.С. Занадворова; Под ред. и с предисл. Е.М. Четыркина. – М.: Финансы и статистика, 1982. – 344 с.

4. *Брюхомицкий Ю.А., Казарин М.Н.* Параметрическое обучение биометрических систем контроля доступа / Вестник компьютерных и информационных технологий. – М.: Изд-во «Машиностроение», 2006. № 2 (20). – С. 6-13.

В.С. Несов, О.Р. Маликов
Россия, г. Москва, ИСП РАН

АВТОМАТИЧЕСКИЙ ПОИСК УЯЗВИМОСТЕЙ В БОЛЬШИХ ПРОГРАММАХ

Введение

В Институте системного программирования РАН была разработана среда обнаружения уязвимостей [1] в исходном коде программ на языке С, позволяющая обнаруживать уязвимости двух типов:

- переполнение буфера (buffer overflow);
- неконтролируемая форматная строка (format string).

Методика обнаружения уязвимостей в данной среде состоит из двух этапов:

- вычисление необходимых для определения уязвимостей атрибутов объектов программы;
- проверка накладываемых на атрибуты условий отсутствия уязвимостей во всех точках программы.

Описанная среда обнаружения уязвимостей [1] имела серьезное ограничение на максимальный размер анализируемых программ. Это было связано с тем, что первоначальная реализация алгоритмов предполагала вычисление и хранение всех атрибутов объектов программы в каждой точке программы. Такой подход позволял проводить анализ программ объемом до 50-60 тысяч строк кода на языке С, используя 2Гб оперативной памяти.

В данной статье описываются:

- способ компактного хранения вычисляемых средой атрибутов;
- механизм подкачки внутренних объектов программы.

Способ компактного хранения вычисляемых атрибутов базируется на том факте, что каждая инструкция программы достаточно ограниченно влияет на проходящий через нее поток данных. Таким образом, в каждой точке программы, вместо хранения всех атрибутов объектов предлагается хранить ссылку на предшествующую точку и накопленные изменения атрибутов. Реализация данного способа компактного представления позволяет снизить расходы среды по памяти примерно в 3.5 раза и по времени в среднем в 1.25 раза.

Разработанный и реализованный механизм сохранения и подкачки вычисляемых атрибутов необходим для снятия ограничений на размер анализируемой программы. Данный механизм позволяет в каждый момент времени хранить атрибуты, вычисленные в пределах только одной текущей функции, плюс значения атрибутов на входах в функции, вызываемые из текущей.

Оставшаяся часть статьи организована следующим образом. В разделе 2 дается краткое описание применяемых в среде [1] алгоритмов, в разделе 3 определяется понятие абстрактной ячейки памяти, которая является основным представлением объектов программы в среде, в разделе 4 приводится описание потока данных в рамках компактного представления, а в разделе 5 описываются алгоритмы работы с определенным в разделе 4 потоком данных. Раздел 6 посвящен описанию алгоритма подкачки внутренних объектов программы в процессе анализа. Результаты экспериментов приведены в разделе 7.

Краткое описание применяемых в среде алгоритмов

Алгоритм, вычисляющий атрибуты объектов программы, является ядром среды обнаружения уязвимостей. В нем можно условно выделить три уровня алгоритмов, работающих на разных фазах анализа. Первый, межпроцедурный уровень – на нем идет работа с отдельными функциями, которые рассматриваются как единое целое. На межпроцедурном уровне вырабатывается метод обхода графа вызовов программы в рамках одной межпроцедурной итерации. Уровень второй, внутрипроцедурный. На нем происходит анализ конкретной функции программы. На внутрипроцедурном уровне также работает итеративный алгоритм, в котором единицами анализа являются отдельные инструкции программы. И, наконец, на нижнем уровне инструкций обнаруживается семантика каждой конкретной инструкции и происходит модификация потока атрибутов, проходящих через данную инструкцию в соответствии с обнаруженной семантикой. После вычисления всех необходимых атрибутов во всех точках программы выполняется проверка необходимых условий отсутствия уязвимостей. Условия накладываются на общее состояние множества атрибутов и зависят от семантики инструкции – точки анализа. Например, для инструкции модификации значения элемента массива условие состоит в том, что в точке входа в инструкцию индекс массива, по которому происходит присваивание, не должен выходить за границы массива.

Абстрактные ячейки памяти

Множество ячеек памяти программы разбивается на идентифицируемые по точке объявления абстрактные ячейки памяти (АЯП) так, что каждые 2 АЯП не описывают одно и то же место в памяти. АЯП сопоставляется объявлениям автоматических переменных функций, объявлениям глобальных переменных, точкам вызова стандартных процедур выделения памяти в куче ([2], sec. 10). Каждой АЯП сопоставляется информация о значении представляемого множества ячеек памяти в каждой точке программы. Для этого при каждом обращении программы к памяти определяется множество АЯП, соответствующее адресам обращения. Так как АЯП сопоставлены непересекающимся областям памяти, вся информация о значениях, к которым происходит обращение, сопоставлена этим АЯП.

Значение АЯП представляет собой набор атрибутов, таких как целочисленный интервал возможных значений, целочисленный интервал возможной длины хранящейся строки (для АЯП, соответствующих массивам символов), признак того, что значение зависит от ввода пользователя, множество объектов, на которые указывает расположенный в АЯП указатель с указанием смещений и т.д.

Представление потока данных

Пусть для графа потока управления $In(I)$ – множество ребер, входящих в вершину I, $Out(I)$ – множество ребер, исходящих из вершины I. Для каждого ребра v графа потока управления определен контекст $C(v)$. Контекстом называется множество пар (АЯП, значение АЯП), определяющих атрибуты АЯП на данном ребре графа потока управления. Каждой вершине I графа потока управления соответствует преобразование потока данных F_I, использующееся при статическом анализе для вычисления контекстов выходных ребер:

$$C(v) = F_I(v, C(In(I))),\ v \in Out(I),\ C(V) = \{C(v), v \in V\}.$$

Пусть для каждой вершины I выделено входящее ребро $P(I) \in In(I)$. Тогда при фиксированных контекстах после применения преобразования F_I для каждого исходящего ребра $v \in Out(I)$ множества

$$\begin{aligned} A(v) &= C(v) \setminus C(P(I)), \\ R(v) &= C(v) \cap C(P(I)) \end{aligned} \tag{1}$$

определяют изменение контекста в вершине I. При этом

$$C(v) = (C(P(I)) \cup A(v)) \setminus R(v). \tag{2}$$

Каждая инструкция программы затрагивает лишь небольшую часть значения контекста, так что размер множеств $A(v)$ и $R(v)$ существенно меньше размера множества $C(v)$. Поэтому для увеличения производительности (за счет избежания копирования значений неиспользуемых АЯП при анализе каждой инструкции) и снижения объема требуемой памяти (за счет избежания хранения копий значений неиспользуемых в данной инструкции АЯП) было применено компактное представление контекстов.

Пусть отношение $P(I)$ определяет дерево (дерево наследования контекстов), покрывающее граф потока управления, так что $P(I)$ – ребро к непосредственному предку вершины I в этом дереве. Компактным представлением контекстов являются множества $A(v)$ и $R(v)$, удовлетворяющие соотношениям (1). При этом значение контекста $C(v)$ для любого ребра v, исходящего из вершины I, определяется при помощи рекуррентного соотношения

$$\begin{aligned} Cr(v) &= (Cr(P(I)) \cup A(v)) \setminus R(v), \\ Cr(v0) &= C(v0), \end{aligned} \tag{3}$$

где $v0$ – входное ребро функции, единственное исходящее из входной вершины графа потока управления, $C(v) = Cr(v)$.

Само значение $C(v)$ при анализе программы не сохраняется.

Так как инструкции требуется получать значения АЯП в контексте входящего ребра, а рекурсивное вычисление контекста каждый раз при обращении занимает много времени, во время анализа строятся временные контексты. Временный контекст $Ct(v)$ ребра графа потока управления v равен контексту этого ребра $C(v)$ и представляется в виде таблицы отображения из АЯП в значения АЯП, так что доступ к временному контексту происходит эффективно.

Алгоритм работы с представлением потока данных

Временный контекст исходящего ребра инструкции получается из временного контекста входящего ребра применением изменений:

$$Ct(v) = (Ct(P(I)) \cup A(v)) \setminus R(v),\ v \in Out(I).$$

Изменения сохраняются в исходящем ребре графа потока управления. При этом измененный временный контекст переносится с входящего ребра на исходящее ребро. При анализе любой вершины графа потока управления создаются временные контексты всех исходящих ребер. Эта процедура требует копирования временного контекста только для инструкций ветвления (имеющих более одного исходящего ребра), доля которых в анализируемых программах достаточно невелика. Псевдокод одной итерации внутрипроцедурного анализа функции приведен на рис.1. Внутрипроцедурный анализ функции выполняется при помощи нескольких обходов графа потока управления, при этом на каждом обходе каждая инструкция посещается один раз. Перед выполнением внутрипроцедурного анализа временные контексты на ребрах графа потока управления отсутствуют для экономии памяти. Перед началом каждого обхода входному ребру функции $v0$ (являющемуся корневым ребром дерева наследования контекстов) сопоставляется временный контекст, равный контексту входа в функцию, $Ct(v0) = C(v0)$. Контекст входа в функцию $C(v0)$ поддерживается постоянно.

Обход организуется в топологическом порядке для дерева наследования контекстов. В топологическом порядке каждая вершина дерева обходится после ее родителя. За счет этого временные контексты спускаются по дереву наследования контекстов и каждый раз при анализе вершины графа потока управления можно гарантировать, что входящему ребру сопоставлен временный контекст, то есть

$$Cr(v) = (Ct(P(I)) \cup A(v)) \setminus R(v),\ v \in Out(I).$$

Порядок обхода графа потока управления выбирается так, что изменения, вносимые анализом инструкций, наиболее эффективно распространяются на инструкции, которые могут читать эти изменения. Для ациклического графа потока управления эффективен любой топологический порядок обхода. Для циклических графов организуется размыкание циклов (удаление части ребер графа), так что при каждом обходе полностью выполняется один проход каждого цикла. Для размыкания циклов применяется интервальный анализ [3]. Пусть G – граф потока управления, Ga – подграф, получаемый из G после размыкания циклов. В качестве дерева наследования контекстов может быть выбрано любое дерево обхода в глубину графа Ga. Топологический обход графа Ga является топологическим обходом такого дерева. Так как при обходе каждая вершина анализируется только один раз, временные контексты всех ее входящих ребер после ее анализа не используются и могут быть удалены. Для каждого ребра графа Ga, вершина, из которой оно исходит, обходится раньше, чем вершина, в которую оно входит. За счет этого по окончании обхода всем таким ребрам не сопоставлен временный контекст.

При анализе вершин слияния потока управления (имеющих более одного входного ребра) на первой итерации внутрипроцедурного анализа некоторые ребра (удаляемые при размыкании циклов) могут не иметь временных контекстов. В этом случае контекст исходящего ребра получается объединением старого значения контекста исходящего ребра и контекстов входящих ребер, имеющих временные контексты.

Однако на всех внутрипроцедурных итерациях, кроме первой, временные контексты всех входящих ребер присутствуют. Ребрам, входящим в Ga, временные контексты сопоставляются при текущем обходе, остальным ребрам временные контексты сопоставляются при предыдущем обходе.

```
// внутрипроцедурная итерация анализа – один обход
// графа потока управления.
intraprocedural_walk(G, F, R, A, v0, C(v0)) {
        // размыкание циклов
        Ga = make_acyclic(G)
        // построение дерева обхода
        P = depth_first_tree(Ga)
        // топологический порядок обхода, S – список вершин
        S = topological_walk(Ga)
        // создание временного контекста в точке входа в функцию
        Ct(v0) = C(v0)
        for(I in S) {
            // если I – вершина слияния потока управления,
            // вызывается apply_join_node
            if (join_node(I))
                apply_join_node(G, I, R, A, P, Ct)
            else
                apply_vertex(G, F, I, R, A, P, Ct)

        }
                                    }
// применение преобразования в вершине
apply_vertex(G, F, I, R, A, P, Ct) {
        for(v in G.Out(I)) {
            // выполнить преобразование
            T = F_I(v, Ct(G.In(I))
            A(v) = T \ Ct(P(I))
            R(v) = T ∩ Ct(P(I))
            Ct(v) = T
        }
        // уничтожить временные контексты входных ребер
        for(v in G.In(I))
            Ct(v) = UNDEF
}
// применение преобразования в вершине слияния потока управления
apply_join_node(G, I, R, A, P, Ct) {
        // ребро O – единственное ребро, содержащееся в G.Out(I)
        {O} = G.Out(I)
        // определить множество входящих ребер, для которых
        // определен временный контекст
        DIn = {v∈G.In(I) | Ct(v) <> UNDEF}
            // в случае если контекст определен не для всех ребер,
        // используется старое значение слитого контекста
        if (DIn <> G.In(I))
            Ct(O) = (Ct(P(I)) ∪ A(O)) \ R(O)
        else
            Ct(O) = ∅
        // применение операции слияния контекстов Join
        for (v in DIn)
            Ct(O) = Join(Ct(O),Ct(v))

        A(O) = Ct(O) \ Ct(P(I))
        R(O) = Ct(O) ∩ Ct(P(I))
        // уничтожить временные контексты входных ребер
        for(v : G.In(I))
            Ct(v) = UNDEF
}
```

Рис. 1. Псевдокод алгоритмов работы с потоком данных

Механизм подкачки

Для снятия ограничений на размер анализируемых программ используется сохранение состояния анализа. Сохраненная информация удаляется из памяти и пере-

стает быть доступна. При необходимости такая информация подгружается обратно в память. Анализируемая программа разбита на модули C/C++. Для каждого модуля поддерживается информация о типах, объявлениях, таблице символов и т.д., используемая при анализе функций, принадлежащих модулю. Глобально (для всего анализируемого пакета) поддерживается граф вызовов. Для межпроцедурного анализа (выбора порядка функций, в котором к ним применяется внутрипроцедурный анализ) требуется только граф вызовов. Для выполнения внутрипроцедурного анализа требуются: граф потока управления; информация, связанная с модулем, которому принадлежит функция; информация о контекстах для анализируемой функции; граничные условия функций, с которыми взаимодействует данная функция.

Взаимодействие происходит при анализе инструкций вызова других функций. Вызывающая функция сообщает вызываемой входной контекст и запрашивает выходной контекст. При вызове передаются значения параметров функции, глобальных переменных и всех достижимых из них по указателям АЯП. Набор требуемых граничных условий заранее (перед непосредственным анализом инструкций вызова) может быть неизвестен, так как функции могут вызываться по указателю, а значение указателя может уточняться по ходу анализа. Для сохранения контекстов используется стандартный механизм языка Java. При сохранении некоторого объекта автоматически сохраняются все объекты, на которые он ссылается. Если некоторый объект в течение сессии сохранения встречается дважды, в файле проставляется ссылка на его образ, созданный при первой попытке сохранения. При считывании в память ссылки между объектами восстанавливаются. Некоторые объекты, сохраняемые на диск, ссылаются на объекты, оставляемые в памяти (и наоборот). Такие ссылки заменяются индексами в таблице, индексирующей целевые объекты. Таблицы хранятся в том же состоянии, что и индексируемые объекты (в памяти либо на диске).

Таблица 1.

Результаты работы среды

	LOC	NOF	TP	Total	Time, sec	Mem, mb	TimeC, sec	MemC, mb	TimeP, sec	MemP, mb
bftpd-1.0.24	3126	114	20	54	30,9	76,2	24,8	18,1	236	1,78
lhttpd-0.1	934	17	5	22	3,6	8	1,7	4,0	6,7	1,87
muh-2.05d	4942	95	12	47	39,2	101,5	30,0	20,8	205	1,54
pgp4pine-1.76	4003	68	16	45	23,7	57,6	24,3	19,1	237	1,91
polymorph-0.4.0	605	15	6	8	2,1	2,8	0,4	3,5	3,4	1,59
popclient-2.21	1701	36	7	34	3,8	7,9	3,6	5,5	25,9	1,87
sharutils-4.2.1	6015	70	11	49	48,3	77,7	39,0	19,3	192	2,02
ssmtp-2.60	2224	33	2	15	8,4	1,8	7,1	7,3	44,9	2,05
surfboard-1.1.8	718	18	16	23	3,8	8,2	1,8	5,2	20,6	2,77
telnetd-1.0	5149	68	3	29	32,4	70,8	32,1	17,5	206	2,66
troll-ftpd-1.26	2354	50	2	35	48,7	51,8	32,2	14,6	187	2,13
Total	31771	584	100	361	244,9	464,3	197,0	134,9	1364,5	22,2

Экспериментальные результаты

В табл.1 показаны результаты анализа 11 пакетов свободно распространяемого ПО. Для каждого пакета приведены его название и номер версии, количество строк исходного кода пакета без учета пустых строк (колонка LOC), количество функций в пакете (колонка NOF), количество истинных предупреждений (колонка

ТР) и общее количество предупреждений (колонка Total). Время работы анализа и объем памяти, занимаемый системой без использования компактного представления, указаны в колонках Time и Mem. Время и объем памяти при использовании компактного представления контекстов представлены в колонках TimeC и MemC. Время и объем памяти при использовании подкачки (допускающей наличие в памяти только одной анализируемой в данный момент функции) и компактного представления контекстов представлены в колонках TimeP и MemP. Сравнение результатов анализа программ различными реализациями среды подтверждает, что реализация способа компактного представления контекстов дает ощутимые результаты. В среднем, время анализа снижается в 3.5 раза, а размер требуемой памяти в 1.25 раза. Результаты, показанные реализацией среды с включенным механизмом подкачки, также крайне интересны и демонстрируют возможность снижения требований по памяти в 60 и более раз даже на сравнительно небольших проектах. Время анализа программ с включенным механизмом подкачки увеличилось в среднем примерно в 7 раз. Несмотря на то, что время анализа увеличилось существенно, можно говорить и о том, что в результате реализации механизма подкачки цель работы была достигнута, были сняты ограничения на размер анализируемых программ.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Маликов О.Р.* Автоматическое обнаружение уязвимостей в исходном коде программ, Известия ТРТУ №4, Материалы VII Международной научно-практической конференции «Информационная безопасность», 2005.- С. 48-53.

2. *Muchnick, Steven S.* Advanced Compiler Design And Implementation, Harcourt Publishers Ltd, Sep. 1997.

3. *Offner, Carl D.* Notes on Graph Algorithms Used in Optimizing Compilers.

4. *Emami M., Ghiya R., and Hendren L.* Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation, Orlando, US, 1994.

5. *Haugh E. and Bishop M.* Testing C Programs for Buffer Overflow Vulnerabilities. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, February 2003.

6. *Wagner D., Foster J. S., Brewer E. A., and Aiken A.* A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In Proceedings of 7th Network and Distributed System Security Symposium, Feb. 2000.

7. *Dor N., Rodeh M., and Sagiv M.* CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pages 155—167, San Diego, California, 2003.

8. *Cousot P. and Cousot R.* Static determination of dynamic properties of programs. In Proceedings of the Second International Symposium on Programming, P.106 - 130. Dunod, Paris, France, 1976.

9. *Bush W.R., Pincus J.D., and Sielaff D.J.* A static analyzer for finding dynamic programming errors. In Proceedings of Software Practice and Experience, P. 775-802, 2000.

Д.И. Морозов
Россия, г. Тюмень, ТГУ

ЭНТРОПИЙНЫЙ МЕТОД АНАЛИЗА АНОМАЛИЙ СЕТЕВОГО ТРАФИКА В IP-СЕТЯХ

Обнаружение в крупных вычислительных сетях, например, вирусных эпидемий на сегодняшний день затруднено, так как наблюдаемые сети обрабатывают большое количество трафика в единицу времени. Во время вирусных эпидемий доля вредоносного трафика составляет малую часть от нормального трафика в подобных сетях. Кроме того, структура вредоносного трафика не известна заблаговременно, что не позволяет использовать сигнатурный метод для его обнаружения