ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ №4(27), 2015, с. 139–156

УДК 519.682.3

В. А. Роганов, А. А. Кузнецов, Г. А. Матвеев, В. И. Осипов

Адаптивный анализ надежности паролей при помощи гибридных суперЭВМ

Аннотация. Статья посвящена вопросам безопасности традиционных, широко распространенных схем работы с паролями.

Показано как современное широкодоступное высокопроизводительное оборудование позволяет проводить эффективные атаки на базы данных с хэшами паролей, которые последнее время все чаще оказываются в руках хакеров. Корнем этой проблемы является устаревшее, легкомысленное отношение к вопросу создания и хранения паролей как системных администраторов, так и конечных пользователей информационных систем.

Авторы статьи не дают полного описания адаптивного алгоритма дешифровки паролей, но делятся некоторыми близкими результатами в надежде на то, что специалисты, отвечающие за хранение и использование паролей, ознакомившись с этой публикацией, смогут сделать свои информационные системы более защищенными.

В статье обосновывается, почему широко укоренившиеся в последние десятилетия в сознании людей схемы создания и хранения паролей следует незамедлительно модифицировать в сторону повышения их надежности.

В заключении статьи даются некоторые конкретные рекомендации в части противодействия интеллектуальному суперкомпьютерному криптоанализу.

Ключевые слова и фразы: T-система, динамическое распараллеливание, язык программирования T++, криптоанализ, информационная безопасность, цепи Маркова, адаптивный подбор паролей.

Введение

Пароли верой и правдой служат делу защиты информации с незапамятных времен. Сегодня подавляющее большинство средств аутентификации в информационных системах базируются на процедуре ввода пароля.

Работы, положенные в основу данной статьи, были выполнены в рамках НИР «Методы и программные средства разработки параллельных приложений и обеспечения функционирования вычислительных комплексов и сетей нового поколения» (№01201354596).

Будучи правильно организованным, этот способ достаточно эффективен: подбор сложного пароля в большинстве случаев оказывается делом малоперспективным в силу как большого пространства для перебора, так и адекватной защитной реакции современных систем по пресечению подобных попыток.

Однако в последние годы один за другим публикуются весьма обширные списки пользовательских паролей от того или иного сетевого сервиса. Ниже приведены наиболее известные случаи такого рода [1].

Летом 2012 года хакеры выложили в открытый доступ 6,5 миллионов паролей от учетных записей социальной сети LinkedIn. Как только информация об этом стала известна, компания извинилась перед пользователями и «сбросила» опубликованные пароли.

В ноябре 2013 года злоумышленники похитили у компании Adobe 150 миллионов паролей. Эта утечка стала рекордной — столько паролей до этого похищать не удавалось никому. Массовая утечка вызвала большой резонанс. В сети даже появился кроссворд из украденных паролей. В ходе разбирательств выяснилось, что для шифрования паролей Adobe использовала довольно простой метод.

В 2014 году список из более чем миллиона паролей пользователей одного известного российского почтового сервиса оказался в публичном доступе. Многие специалисты сходятся во мнении, что подобный масштаб указывает на дешифровку паролей из украденной базы данных с использованием суперкомпьютера.

Хотя информация о каждой конкретной утечке может быть недостоверна, общий вывод таков: ситуация в области защиты паролей стала критической. Чтобы разобраться в причинах этого явления, следует рассмотреть традиционные схемы работы с паролями. Если затем принять во внимание тот факт, что мощности современных компьютеров выросли на порядки с тех пор, когда схемы шифрования паролей были придуманы, а также учесть достижения современного интеллектуального криптоанализа, то станет совершенно ясно, что схемы создания и хранения паролей необходимо срочно и решительно совершенствовать.

1. Традиционные схемы работы с паролями

Для установления подлинности пользователя в процессе его аутентификации информационная система должна проверить введенный им пароль. На заре появления многопользовательских компьютерных

систем пароли хранились в незашифрованном виде в специальных защищенных от чтения аккаунт-файлах; считалось, что вся необходимая защита паролей обеспечивается на уровне файловой системы.

Однако сравнительно быстро выяснилось, что уповать исключительно на отсутствие прав доступа к аккаунт-файлам очень недальновидно, и был сделан фундаментальный шаг в сторону повышения безопасности: пароли стали хранить в зашифрованном виде. Процедура проверки подлинности при этом меняется незначительно: вместо сравнения введенного пользователем пароля с оригиналом система вначале шифрует введенный пароль и уже затем сравнивает его с зашифрованным образцом – хэш-функцией от пароля.

Поначалу все были так уверены в надежности нового подхода, что в системе Unix зашифрованные пароли хранились в открытом для чтения файле /etc/passwd. Эйфория улетучилась после того, как вначале у спецслужб, а затем и у продвинутых хакеров появились программы, способные дешифровать значительную часть хранимых таким образом паролей в течении нескольких минут.

Авторы помнят момент, когда в одном из ВУЗов системными программистами была проведена работа по переделке схемы хранения паролей. К сожалению, инженеры там не были осведомлены о методах дифференциального криптоанализа, и использовали слишком уж простой способ шифрования паролей. Пароль разбивался на три части $\{\mathbf{X}, \mathbf{Y}, \mathbf{Z}\}$, которые преобразовывались в три новых числа $\{(\mathbf{X}^*\mathbf{Y})\hat{}\mathbf{Z}, (\mathbf{Y}^*\mathbf{Z})\hat{}\mathbf{X}, (\mathbf{Z}^*\mathbf{X})\hat{}\mathbf{Y}\}$. Решение подобных нелинейных систем уравнений даже на компьютерах тридцатилетней давности занимало считанные секунды, и мы оставляем дешифровку зашифрованных таким способом паролей пытливым читателям в качестве упражнения.

Следующей историей, взбудоражившей весь мир, была публикация метода взлома паролей при помощи так называемых «радужных таблиц» [3]. Это было применение достаточно простой и хорошо известной идеи, которая удивительно хорошо подошла для дешифровки паролей, зашифрованных при помощи криптостойких однонаправленных хэш-функций.

Наиболее интересным моментом в технологии радужных таблиц является тот факт, что они составлялись при помощи больших вычислительных комплексов неделями, а их применение (дешифровка паролей) занимало считанные минуты или даже секунды. Ситуация

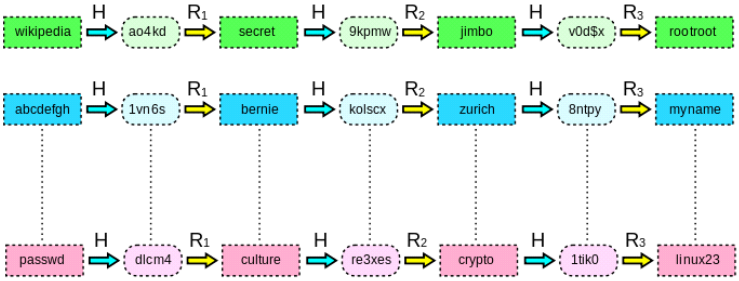

Рис. 1. Схема радужной таблицы

усугублялась тем, что многие сетевые сервисы (включая Microsoft LAN Manager, а также значительный процент Web-сайтов) оказались практически беззащитными перед этой новой атакой: перехваченные пароли могли расшифровываться хакерами практически в реальном времени.

Для составления радужных таблиц наряду с используемой известной хэш-функцией $\mathbf{H}$ используются дополнительные функции редукции $\mathbf{R_i}$, которые служат для обратного отображения хэшей на множество паролей; с их помощью строятся так называемые цепочки хэшей.

Из состава каждой такой цепочки запоминаются только первый и последний элементы, что позволяет избежать проблем с размером таблицы. При восстановлении пароля по значению хэш-функции требуется всего лишь построить несколько возможных цепочек хэшей по тому же самому принципу, и в случае обнаружения последнего элемента цепочки перейти в ее начало, что с хорошей вероятностью позволяет «добежать» до интересующего нас пароля.

Бороться с применением радужных таблиц довольно просто: достаточно в каждой информационной системе использовать индивидуально-модифицированную версию криптостойкой хэш-функции. В простейшем случае использовался «сдвиг» аргумента-пароля путем его склеивания с так называемой «солью». «Соль» выбирается для каждой системы некоторым случайным образом, что делает построение универсальной радужной таблицы невозможным.

К сожалению, история о радужных таблицах надолго всех успокоила: после нее широко распространилось мнение от том, что достаточно

использовать «соль», и это обеспечит защиту от дешифровки паролей привычной длины. В следующем разделе мы покажем, почему сегодня это уже не так.

2. О правилах составления паролей и о персональных суперЭВМ

Перед тем, как перейти к основному тезису этой статьи, хотелось бы кратко описать еще одну историю, связанную с анализом защищенности современных Интернет-сайтов. Из нее становится понятно, что степень защиты паролей при помощи вышеописанной традиционной схемы сегодня уже не может рассматриваться как приемлемая.

Сегодня многие сайты создаются на основе широко распространенных CMS (Content Management System). В нашей стране, например, особой популярностью пользуется CMS Joomla. Разработанные профессиональными программистами ядра подобных систем могут считаться условно надежными, однако реальный функционал каждого конкретного сайта обеспечивается дополнительными модулями-расширениями. Последние создаются уже гораздо менее квалифицированными разработчиками, и нередко содержат в своем коде уязвимости, например, типа «SQL-инъекция». Следствием эксплуатации такого рода уязвимостей является утечка значений хэш-функции от хранящихся пользовательских паролей. А поскольку в системе Joomla хэши вычисляются при помощи стандартной функции MD5 и хранятся в базе данных вместе с «солью», то все данные для дешифровки паролей оказываются после этого в руках атакующих. Если пароли у части пользователей оказываются простыми (например, достаточно короткими), то использование стандартных утилит типа hashcat [4] позволяет достаточно быстро провести их дешифровку.

Для того, чтобы совершить этот последний шаг — дешифровку хэшей, достаточно перебрать множество наиболее вероятных паролей и вычислить соответствующие значения хэш-функции. Вот здесь-то и срабатывает стереотип, сложившийся в предыдущие десятилетия: люди склонны считать, что имеющихся у частных лиц вычислительных ресурсов для эффективного проведения такого перебора недостаточно.

Грубую оценку количества паролей в так называемом «хорошем словаре», то есть наборе последовательностей символов, который содержит значительную долю пользовательских паролей типичной длины (с ограничением порядка 12-ти символов), можно получить,

исходя из количества исходных «идей» для построения пароля (то есть, типичных слов языка и чисел) и вариантов их возможного комбинирования/видоизменения с учетом ограничения на суммарную длину. Если считать, что количество базовых слов исчисляется десятками тысяч, а типичный пароль составляется не более чем из двух слов и «разбавляется» не вполне случайными цифрами по достаточно типичным правилам, то можно заключить, что в нашем «хорошем словаре» окажется порядка триллиона таких «традиционных» паролей.

Теперь оценим общее количество времени, которое потребуется для перебора такого количества вариантов. Для случая широко распространенной хэш-функции MD5 проверка одного пароля требует нескольких тысяч примитивных арифметических операций. Поскольку процесс дешифровки множества паролей хорошо распараллеливается, а современный видеоадаптер содержит порядка тысячи процессорных ядер, работающих на тактовой частоте порядка гигагерца, нетрудно прийти к выводу, что за секунду счета теоретически можно перебрать порядка одного миллиарда паролей. То есть, современный видеоадаптер способен перебрать пароли из нашего «хорошего словаря» в течение часа.

Конечно, эти упрощенные оценки можно подвергнуть жесткой критике. Во-первых, мы не учли того факта, что процесс составления пароля из базовых слов и правил все же достаточно интеллектуален, а видеоадаптеры эффективны лишь для множества однотипных, векторных вычислений. Во-вторых, заявленную нами оценку объема «хорошего словаря» в один триллион паролей легко обойти, используя чуть более длинные пароли, которые следует более щедро разбавлять цифрами и «приправлять» изменениями в регистре употребляемых букв.

К сожалению, подобные замечания ничего радикально не меняют. В распоряжении злоумышленников зачастую имеется вовсе не одиночный видеоадаптер, а тысячи компьютеров с видеоадаптерами, входящими в состав подчиненной им сети. Поэтому верхнюю оценку для размера типичного «хорошего словаря» можно смело поднять до 10^{15} записей, что соответствует, по ряду наших оценок, паролям, содержащим уже порядка 15-ти символов. В отношении же динамического построения и векторизации вычислений хэш-функции от множества паролей ситуация следующая: современная аппаратура и системное программное обеспечение позволяют без труда векторизовать динамическое построение словаря по достаточно сложным правилам, что мы

и покажем в следующим разделе на примере криптоаналитического приложения md5t.

3. О современных гибридных суперЭВМ, о возможностях технологии CUDA и динамическом распараллеливании вычислений

Сегодня распространено мнение, что публичное обсуждение и публикация научных результатов по вопросам информационной безопасности в итоге служит на пользу как защите информационных систем, так и информационного общества в целом. Мы приведем некоторые алгоритмы и фрагменты кода в качестве демонстрации того, насколько просто и дешево может быть осуществлена интеллектуальная дешифровка паролей при помощи современных технологий.

Еще совсем недавно под суперЭВМ понимались установки с производительностью порядка одного терафлопса, или 10^{12} примитивных арифметических операций в секунду. Сегодня же одна топовая видеокарта от NVidia обеспечивает производительность, примерно на порядок превышающую эту величину. Это достигается с использованием векторной архитектуры, или, более точно, архитектуры SIMT (Single Instruction Multiple Threads).

Создание программ для векторных устройств сопряжено с трудностями процесса *векторизации*, или процесса представления алгоритмов в виде, когда значительное количество параллельных потоков в большинство моментов времени выполняют одну и ту же инструкцию.

В нашем случае при дешифровке паролей есть два вида вычислительной работы, которую желательно переложить на графический ускоритель общего назначения (GPGPU): динамическое создание паролей (статически создавать массив в триллион паролей непрактично), а также их обработку (вычисление значений хэш-функций). Для адаптивной дешифровки паролей эти две стадии желательно максимально сблизить: в случае успехов/неудач в одной из ветвей перебора можно оперативно информировать другие ветви о направлениях наиболее перспективного набора правил конструирования; словарь в этом случае может динамически расширяться и сужаться в зависимости от набора уже расшифрованных паролей. Иными словами, выгодно поместить всю логику интеллектуального криптоанализа *внутрь* GPU, чтобы взаимодействие с центральным процессором не являлось узким местом, тормозящим процесс перебора.

До недавнего времени это было трудно реализовать, однако в последние годы технология CUDA начала поддерживать динамическое распараллеливание. Это позволило реализовать подмножество языка T++ для относительно комфортного программирования подобного рода программ для GPU [5].

Экспериментальное приложение md5t работает следующим образом. Все слова «хорошего словаря», длиной до 10-ти символов, разбиваются на три сегмента:

111222[3333]

Символы третьей группы заключены в квадратные скобки в знак того, что их количество и число вариантов непостоянно и определяется символами предыдущих двух групп.

Для перебора каждой возможной первой тройки символов вызывается обычная T-функция g(), которая маршрутизируется на наименее загруженный узел гибридной суперЭВМ (вычислительного кластера с видеоадаптерами) обычным для T-системы образом:

T++

```
tfun int main(int argc, char *argv[]) {
    proxyAccelConf(new MData());
    double found = 0;
    tval double r[N*N*N];
    unsigned i = 0;
    s3_t s1;
    for_each_s3(s1, { ts3_t ts1; ts1=s1; r[i++] = g(ts1); })
    do {
        found += r[--i];
    } while(i);
    fprintf(stderr,"found %lf passwords",found);
    return 0;
    }
```

В теле T-функции g() происходит вызов другой T-функции, выполняющейся уже на GPU. Эта функция без обращения к центральному процессору перебирает все возможные комбинации символов второй тройки и динамически инициирует запуск низкоуровневого, эффективно векторизованного ядра (функции f2()), для перебора оставшихся вариантов символов третьей группы. Количество вариантов на третьем уровне перебора определяется первыми шестью символами и может колебаться в весьма широких пределах в соответствии с правилами конструирования «хорошего словаря», о котором мы подробнее

расскажем ниже.

Хотя код для T-функций, исполняющихся внутри GPU, обладает некоторой спецификой, по сути он следует функциональному стилю T-системы. Отличия обусловлены в-основном желанием максимально задействовать низкоуровневые возможности при кодировании листовых функций, а также тем, что многие атрибуты T-функций для GPU пока не имеют представления на уровне конвертера языка T++. Тем не менее, данный код может одновременно выполняться как на GPU, так и на CPU, что достигается путем его раздельной компиляции и компоновки в разных пространствах имен и макроопределениях. Практически это проявляется в том, что если на компьютере не установлен GPU с поддержкой CUDA, все вычисления будут производиться на центральном процессоре.

T++ –

```
struct f2: public TFun<double> {
    s6_t s12;
    DEV f2(long s1)
    {
        memcpy(s12,&s1,3);
        needThreads(1);
    }
    DEV double body()
    {
            double found = 0;
        TPtr<double>* p = new TPtr<double>[N*N*N];
        unsigned i = 0;
        f3::jb_t b;
        for_each_s3(s12+3, {
                  memcpy(b.s12,s12,6);
                b.m = MONEY -md->price6(s12);
                unsigned ei = s12[5]-'a';
                b.p = md->tab[ei][b.m];
                if (b.p[0] >= THBLK/2)
                        p[i++] = ACCEL_TCALL(f3,((long)&b));
        });
            do {
                    found += *(p[--i]);
        } while(i);
        delete [] p;
        return found;
    }
    };
```

Функция f2() инициирует перебор третьего уровня, вызывая вектори-

зованную низкоуровневую T-функцию f3(), код которой представлен ниже. В качестве аргумента ей передается указатель на структуру, содержащую элементы третьего уровня (последние возможные символы), которые имеет смысл перебирать.

T++ –

```
struct f3 : public TFun<double> {
    struct jb_t {
        s6_t s12;
        money_t m;
        unsigned *p;
    } b;
    bool ok[THBLK];
    DEV f3(long pb) : b(*(jb_t*)pb)
    {
        memset(ok,0,sizeof(ok));
        needThreads(THBLK);
    }
    DEV void llBody(int id, double *found)
    {
        if (found==NULL) {
            char s[11] = {0,};
            unsigned cnt = b.p[1]; //aligned cnt
            assert(!(cnt % THBLK));
            unsigned todo = cnt/THBLK;
            unsigned *p = b.p + 2; //skip r&a cnts
            for(int k=0; k<6; k++) s[k]=b.s12[k];
            for (unsigned i=0; i<todo; i++) {
                uchar* sfx=(uchar*)(p+id*todo+i);
                for(int k=0; k<4; k++) s[k+6] = sfx[k];
                unsigned len=0;
                for (int k=0; k<10; k++) len+=(((char*)s)[len]!=0);
                uint32_t h[4] = { -1u, };
                md5((uint8_t*)s,len,h);
                int pi = md->hcmp(h); //password index
                if (!(pi<0)) {
                    printf("PASSWORD[%d]: %s",pi,s);
                    ok[id] = true;
                }
            }
        } else {
            *found += ok[id]?1.0:0.0;
        }
    }
    };
```

В теле этой функции вызывается функция MD5, линейный код которой

задан при помощи набора макроопределений.

Вычислительная тяжесть ядра функции третьего уровня может существенно меняться; динамическая балансировка и планирование листовых T-функций обеспечивается встроенным планировщиком и низкоуровневыми возможностями CUDA Dynamic Parallelism с участием ядра T-системы (управление очередью задач и синхронизация по данным между асинхронными вызовами).

В завершение мы дадим описание алгоритма для оценки размера «хорошего словаря». Такая оценка необходима, поскольку вычислительные ресурсы в любом случае ограничены. Эффективный алгоритм оценки размера без вычисления всех элементов словаря позволяет проводить эксперименты с набором правил, и, в случае адаптивного поведения программы, на ходу модифицировать используемый набор правил при конструировании паролей.

4. Марковские цепи и оценка размера «хорошего словаря»

При создании экспериментального приложения md5t был использован относительно простой способ построения «хорошего словаря».

Основная идея этого способа состоит в том, что слова в большинстве естественных языков представляют собой далеко не случайные цепочки символов и строятся из слогов. При этом вероятности тех или иных возможных буквосочетаний могут весьма существенно различаться.

Для ограничения размера словаря введем понятие цены слова. Слово будет тем дороже, чем больше редких сочетаний букв в нем встречаются. То есть, чем более популярны составные части слова, тем оно для нас дешевле. «Хорошим словарем» будем называть множество слов, цена каждого из которых не превышает некоторой фиксированной константы.

Если посмотреть на такой словарь с точки зрения его случайного перебора, то можно сказать, что выбор следующей буквы будет тем вероятнее, чем ниже цена получающегося при ее добавлении слова и чем больше типовых буквосочетаний можно образовать при последующем добавлении букв. Это свойство цепочек символов напоминает свойства так называемых цепей Маркова [**6**].

Для оценки размера заданного таким образом словаря удобно использовать принцип динамического программирования. Ниже приведены отдельные фрагменты кода на языке Haskell с пояснениями,

которые демонстрируют использованный метод оценки общего размера словаря.

Haskell

```haskell
-- Нас будут интересовать буквосочетания длиной от двух до трех букв.

type Chunk = [Char]
minChunk = 2
maxChunk = 3

-- Добавление очередной буквы образует новые буквосочетания.

newChunks:: [Char] -> Char -> [Chunk]
newChunks rs c = concat $ map f [minChunk..maxChunk] where
    f k = [reverse $ take k $ c:rs | k <= length (c:rs)]

-- Перечисление всех буквосочетаний в данном слове.

chunks:: String -> [Chunk]
chunks = f [] where
   f _ [] = []
   f rs (c:tl) = newChunks rs c ++ f (c:rs) tl

-- Цена слов измеряется целыми числами.

type Money = Int
type Price = Int

-- Цену одного буквосочетания назначим в зависимости от того,
-- сколько раз оно встречается в образце текста,
-- на котором обучается программа.

chunkPrice:: Chunk -> Price
chunkPrice = let
  ht = (unsafePerformIO $ HT.fromList al)::
HT.CuckooHashTable Chunk Int
  al = cnt $ sort $ chunks sample
  cnt [] = []
  cnt (e:tl) = (e,1+length es): cnt nonEs where
        (es,nonEs) = span (==e) tl
    in
    \c -> unsafePerformIO (HT.lookup ht c >>= return. f) where
          f Nothing = 2
          f (Just _) = 1

-- Будем составлять таблицу, содержащую количество (длинное число Integer)
-- цепочек с данной ценой и суффиксом в соответствии с принципом
```

```
-- динамического программирования: построение следующих уровней в таблице
-- опирается на вычисленные на предыдущих уровнях значения.

type NthLevel = [((Chunk,Price),Integer)] -- sorted!
levelSize:: NthLevel -> Integer
levelSize = sum. map snd

level0 = [(("",0),1)] -- zero level => single empty zero-
priced chunk
nextLevel:: NthLevel -> NthLevel
nextLevel prev = filter canBuy $ merge $ sort l where
  canBuy = (totalMoney >=).snd.fst
  l = [((take (maxChunk-1) (c:rs), p+addon rs c), n) |
                ((rs,p),n) <-prev, c <-charSet ]
  merge ((e,n):(e',n'):tl) | e==e'= merge $ (e,n+n'):tl
                           | otherwise = (e,n): (merge $ (e',n'):tl)
  merge l = l

addon rs = sum. map chunkPrice. newChunks rs
charSet = ['a'..'z']
totalMoney = 21
```

При построении таблицы со значениями количества цепочек для слов заданной цены и заданным последним суффиксом-буквосочетанием мы используем рекурсию и заглядываем на предыдущий уровень. Таким образом мы быстро оцениваем количество слов, цена которых не превышает заданный лимит totalMoney, не перебирая при этом непосредственно сами слова.

Заключение

Вряд ли следует как-то обосновывать утверждение о том, что безопасность систем ухудшается, если третьи лица тем или иным способом получают доступ к косвенным сведениям о паролях. К такого рода сведениям относятся:

- код хэш-функций и значения хэш-функций от паролей;
- устаревшие пароли, а также пароли тех же пользователей на других ресурсах;
- традиционные правила составления паролей.

Информация первого рода появляется у злоумышленников вследствие разного рода утечек, а также в случае, когда информация о хэшах паролей передается по сети в незашифрованном виде.

Информация второго рода сознательно аккумулируется хакерами, которые целенаправленно занимаются сбором и анализом информации о паролях.

Наконец, третья группа сведений, или формализованный свод правил, по которым многие пользователи наиболее часто составляют свои пароли, и вовсе общеизвестна. Кроме того, набор правил непрерывно пополняется и корректируется для различных групп пользователей по мере анализа устаревших и успешно дешифрованных паролей.

Все вышеперечисленные моменты являются звеньями одной цепи, которая при темпах сегодняшнего развития компьютеров неминуемо приведет к необходимости в корне изменить привычные для всех принципы составления и использования паролей. В противном случае степень защищенности среднестатистического пользовательского пароля не позволит серьезно говорить о какой либо защите в принципе.

Актуальность данной проблемы становится критической именно сегодня, когда в руках рядового хакера оказываются спецвычислители на базе графических процессоров (технологии CUDA и OpenCL), суммарная вычислительная производительность которых исчисляется десятками триллионов операций в секунду. В совокупности с неуклонно развивающимся ПО для интеллектуального криптоанализа это способно породить серьезный вызов информационному сообществу.

Поэтому именно сегодня надо срочно менять «традиции» составления и хранения паролей. Ниже даны некоторые рекомендации, которые значительно затрудняют применение вышеописанных методов дешифровки паролей.

- Короткие пароли необходимо полностью исключить из употребления.
- Современные информационные системы должны позволять использовать длинные пароли, желательно не менее 25-ти символов.
- Пароли, построенные по простым правилам, необходимо распознавать и безоговорочно отклонять. Информационные системы должны проводить анализ паролей при их создании и изменении. В том случае, если пароль попадает в тот или иной «хороший словарь», пользователь должен получать уведомление о необходимости срочно сменить пароль.
- В информационном обществе необходимо вести разъяснительную работу о важности использования действительно оригинальных, сложно-составных паролей. Необходимо объяснять необходимость

иметь различные пароли для разных сервисов и избегать заимствования способов конструирования паролей у друзей и знакомых.

- В качестве основы для шифрования паролей следует использовать тяжелые в вычислительном смысле хэш-функции.

Разработчики хэш-функций обычно гордятся их эффективностью. В данном случае это серьезный недостаток, а не достоинство.

Именно высокая скорость вычисления хэш-функции позволяет эффективно перебирать квадриллионы паролей. Хэш-функции можно утяжелять либо путем их нерегулярной итерации, либо в цикле вычислять множество хэш-функций для разных значений «соли» с последующим их объединением.

- Для борьбы против применения векторных вычислителей можно использовать оригинальные хэш-функции, в коде которых много условных операторов и ветвлений, зависящих от символов пароля.

Такую хэш-функцию достаточно просто вычислять на CPU, но реализовать ее эффективное параллельное вычисление на GPU и прочих векторных вычислителях для массива хэшей будет уже не так просто.

- При авторизации следует по возможности избегать передачи зашифрованных паролей по сети в явном виде.

Хотя логически такая передача необходима, физически ее следует осуществлять в рамках защищенного соединения, например, с использованием безопасных протоколов типа HTTPS. Само же хранилище паролей следует защищать значительно серьезнее, чем это делается сегодня.

Список литературы

[1] *Входите, открыто: история крупнейших утечек паролей*, URL http://siliconrus.com/2014/09/opened ↑ 140.

[2] *Экспресс-проверка паролей на надежность*, Лаборатория Касперского, URL https://blog.kaspersky.ru/password-check/ ↑ .

[3] *Rainbow table*, URL https://en.wikipedia.org/wiki/Rainbow_table ↑ 141.

[4] *OclHashCat: GPGPU-based multi-hash cracker using a brute-force attack*, URL http://hashcat.net/oclhashcat/ ↑ 143.

[5] В. А. Роганов, А. А. Кузнецов, Г. А. Матвеев, В. И. Осипов. «Реализация Т-системы с открытой архитектурой для CUDAустройств с поддержкой динамического параллелизма и для гибридных суперЭВМ на их основе», *Программные системы: теория и приложения*, **6**:1(24) (2015),

с. 175–188, URL http://psta.psiras.ru/read/psta2015_1_175-188.pdf ↑ 146.

[6] *Цепь Маркова*, URL https://en.wikipedia.org/wiki/Markov_chain ↑ 149.

Рекомендовал к публикации *д.ф.-м.н. С. В. Знаменский*

Об авторах:

Владимир Александрович Роганов

Научный сотрудник ИПС им. А.К. Айламазяна РАН. Разработчик современных версий Т-системы, ведущий разработчик системы «OpenTS». Принимал активное участие в суперкомпьютерных проектах Союзного государства России и Беларуси, в том числе в проектах «СКИФ» и «СКИФ-ГРИД».

e-mail: var@pereslavl.ru

Антон Александрович Кузнецов

Научный сотрудник ИПС им. А.К. Айламазяна РАН, разработчик системного и прикладного ПО. Один из разработчиков системы параллельного программирования «OpenTS». Область научных интересов: параллельное программирование, компиляторы, распределенные вычисления в гетерогенных средах, геоинформационные системы.

e-mail: tonic@pereslavl.ru

Герман Анатольевич Матвеев

Ведущий инженер-исследователь ИЦМС ИПС им. А.К. Айламазяна РАН. Один из разработчиков системы «OpenTS». Принимал участие в суперкомпьютерных проектах Союзного государства России и Беларуси.

e-mail: gera@prime.botik.ru

Валерий Иванович Осипов

К.ф.-м.н., научный сотрудник ИПС им. А.К. Айламазяна РАН. Один из разработчиков системы «OpenTS». Принимал участие в суперкомпьютерных проектах Союзного государства России и Беларуси.

e-mail: val@pereslavl.ru

Пример ссылки на эту публикацию:

В. А. Роганов, А. А. Кузнецов, Г. А. Матвеев, В. И. Осипов. «Адаптивный анализ надежности паролей при помощи гибридных суперЭВМ», *Программные системы: теория и приложения*, 2015, **6**:4(27), с. 139–156.
URL http://psta.psiras.ru/read/psta2015_4_139-156.pdf

Vladimir Roganov, Anton Kuznetsov, German Matveyev, Valeriy Osipov. *Adaptive analysis of passwords' reliability using computational power of hybrid supercomputers.*

Abstract. The article concerns the security of traditional ways of password storing and handling. It describes known incidents of compromising password hash databases using high-performance computing clusters. It is noted that very often the cause for database vulnerabilities is an obsolete way of storing and handling passwords by users and system administrators. The research shows that adaptive cryptanalysis algorithms introduce the considerable security threat for password-storing information systems. Authors give security recommendations so as to help prevent intelligent cryptanalysis performed on HPC hybrid cluster systems. (*In Russian*).

Key Words and Phrases: T-system, dynamic computing, T++ programming language, cryptanalysis, information security, Markov chains, adaptive password searching, HPC.

References

[1] *Vkhodite, otkryto: istoriya krupneyshikh utechek paroley*, URL http://siliconrus.com/2014/09/opened.

[2] *Ekspress-proverka paroley na nadezhnost'*, Laboratoriya Kasperskogo, URL https://blog.kaspersky.ru/password-check/.

[3] *Rainbow table*, URL https://en.wikipedia.org/wiki/Rainbow_table.

[4] *OclHashCat: GPGPU-based multi-hash cracker using a brute-force attack*, URL http://hashcat.net/oclhashcat/.

[5] V. A. Roganov, A. A. Kuznetsov, G. Matveev, V. I. Osipov. "Implementation of T-system with an open architecture for CUDA devices supporting dynamic parallelism and for hybrid computing clusters", *Programmnye Sistemy: Teoriya i Prilozheniya*, **6**:1 (2015), pp. 175–188 (in Russian), URL http://psta.psiras.ru/read/psta2015_1_175-188.pdf.

[6] *Markov chain*, URL https://en.wikipedia.org/wiki/Markov_chain.

Sample citation of this publication:

Vladimir Roganov, Anton Kuznetsov, German Matveyev, Valeriy Osipov. "Adaptive analysis of passwords' reliability using computational power of hybrid supercomputers", *Program systems: theory and applications*, 2015, **6**:4(27), pp. 139–156. (*In Russian*). URL http://psta.psiras.ru/read/psta2015_4_139-156.pdf