

想成为一名破解者吗? [2006-12-13 更新完成至 Part III]

本系列文章由 RednaxelaFX 翻译自 insani 在他们的网站上提供的破解相关教程,面向的读者群是有志于在中文化 GALGAME 中程序方面的工作贡献力量,但是尚未入门的人.文章是以原作者的视角,以第一人称展开的,与读者交谈式的破解教程.到目前(2006-12-03)为止已发表的有四篇,主要都是描述资源文件的处理相关的技术的,有些部分或许会与译者之前发过的帖有所重叠.今后或许也会涉及到游戏原程序(不是源程序=_=)的修改相关,若出现的话或许也会翻译出来与大家一同学习.希望该系列的文章能对中文化工作者在程序技术上有所帮助.对程序处理方面已经具备相当经验的人来说本系列文章多是十分基础的,请快速跳过已经熟知的部分.当然其中要是翻译错误或者原文错误的话,敬请指正 ^ ^

原文已发表的篇章:

[So You Want To Be a Hacker?](#)

[Part I: Abilities and Responsibilities](#)

[Part II: The Hex Editor](#)

[Part III: Code Prototyping](#)

[Part IV: Compression Formats](#)

原文可以在 <http://sekai.insani.org/> 找到,原作者 Edward Keyes.同一站上也有关注翻译者的相关技能的系列文章《Letters to a Young Translator》,作者 Seung Park.

关于 insani 及其相关作品,请到 <http://www.insani.org/> 查看.他们是一个二人组合,虽然两人都有英化作品所需的全方位能力,却还是分工为各自分别专门负责程序和翻译的进行.值得一提的是,较流行的英文版 ONScripter 就是 insani 改造版.原作者的工作环境并不总在 Windows 下,因而所用工具也并非仅限于 Windows 平台上.

本文的翻译保留了原文的第一人称形式;而对原文稍微调整的部分,纯属个人见解,无意歪曲原作者的意思.特此说明.

其实是想说...与其说是翻译,还不如说做成"读书笔记"了...总之蓝色的部分是翻译自原文的就是了 ^ ^

想成为一名破解者吗? Part I: 所需技能与相应责任

(译者按: Part I 的开始前想先提一下 insani 文章的风格.他们在系列文章的开篇总会提到"责任"问题.这部分打算按照原文翻译,但并不代表译者完全认同其观点.这部分的翻译之后,也会附上一段 Susie 的简单使用介绍.另外对"hacker"一词翻译为了"破解者",因为译者认为实际上文章所描述的工作与其说是 hack 还不如说是 crack.)

"我想翻译游戏,但是我一点日语都不懂." 如果每次听到这种话我都用一个字节来记录的话,我可以换掉我的硬盘了...

所幸的是,在做翻译之外还有一条路——这与[捣弄假名然后为代词都跑哪去了而疑惑]需要的是完全不同的一套技能.然而这项工作与翻译同等重要.试想没有你提供的脚本的话,一个翻译能对着什么翻译? 没有你提取出的图像,一个改图者能改什么图? 没有你重新组装好的游戏,一个玩家能玩什么游戏?

所以,问题就变成了,"想成为一名破解者吗?"

或者,抓住要点来说,你拥有破解游戏所需要的技能吗? 破解者(或者程序员,或者技术指导,或者随便怎么称呼)在翻译过程中的早期阶段有着重要的职责.最优先的就是找出提取游戏数据的方法,具体来说就是脚本和图像.这样小组的其他成员才能开始工作.其次,他必须找到把修改过的数据重新放入游戏中而不出错的方法.最后也是最麻烦的,他经常须要修改游戏本身的方法来让游戏正确使用修改过的资源(例如,将游戏的文本引擎改为使用半角英语字符,或者实现日语游戏所不需要的自动换行功能)

(译者按: 对于中文化工作者来说,这些例子的对应版本就会是修改字体的参数来让汉字编码得以正确显示,或者修改边界值来逃过字符集的边界检查等.上面提到的自动换行原文是 `word-wrapping`,关键在于自动换行时要考虑到单词的完整性.)

提取过程有时候很简单,因为在对 **H CG** 的不懈追求中,可能已经有其他破解者写好了从游戏提取图像的工具.相比之下,再插入工具则甚为罕见;你几乎总是得自己来写,除非你处理的游戏与其他的翻译小组以前处理过的使用了相同的游戏引擎.就算顺利提取出了脚本,在把脚本交给翻译前,你得先想清楚他会返回些什么东西...你能把脚本自动再插入,还是得自己手动复制粘贴一行行的脚本呢?

有一点我要明确一下,我不会单纯为了盗版而讨论破解防拷贝技术.这里所讨论的破解纯粹是为了翻译自由发行游戏,或者是为已经购买了完整的日语版游戏的人制作英语补丁.

(译者按: 这点翻译得相当无奈.在一些客观条件的限制下,玩家的需求与条件的限制之间的矛盾,促使越来越多的中文化工作组出现和成长.相信多数的非官方中文化工作者,在中文化一部游戏作品的时候,并不是站在侵犯任何组织或个人的权益的出发点上的.相反,这是在得

不到有效的官方代理的条件下,为了推广自己喜欢作品而尽的努力.在可能的范围内,中文化工作者都应该尽量少涉及与作品原作者权益相背的范围.但是许多时候实在是事出无奈...嗯,这个问题就此打住了.Ed 的观点译者在此只能理解却无法认同了.)

你需要已经具备哪些知识呢?如果你在考虑这个问题,我希望你懂得一定的编程,因为那个我教不了你.好在,你并不会经常需要“大量”编程...只要些基本工具就够用了,越底层越好:你连如何在 Windows 中制作一个单选按钮都不需要知道,但是你得知道区分 big 与 little-endian.这个,我能教会你.

来想想所有你懂得的程序设计语言.把不能轻松完成如在文件与未处理的字节数组间读写之类的一些底层操作的语言扔掉.用你觉得最顺手的语言,或者也可以尝试点新东西.个人而言,我主要直接用 C,不过如果我得从头再做一次的话,我可能会仔细考虑一下,比如说, Python, Java, C++,或者其它好用常用的语言.不过如果你最喜欢的语言是 Visual Basic 的话...我觉得你是时候扩展一下你的技能了.

(译者按: 嗯,这部分译这也想说点.在中文化游戏的过程中,破解者自己所写的程序多数只是自己临时使用的,所以就算慢一点或者代码乱一点问题也不大.在语言的选择范围上也就有相当大的余地.如 Ed 所说,用什么语言其实是自己的习惯,并不一定要专门另外去学一门困难的语言的.然而如果完全不懂得编程的话而又想加入到中文化工作的程序破解方面中,如果不学习哪怕一点点的编程知识,会带来比较大的限制.

对于有 C 或者语法与 C 类似的(例如 C++, Java, C#)语言基础的人而言,安心继续使用你觉得顺手的就可以了.熟悉 JavaScript 或者 Flash ActionScript 的,在破解时也可以考虑转向 Java 或者 C#,语法比较相似.haeleth 喜欢用 CAML,ML 的一种变体,那个也是相当方便的语言.

如果是完全没接触过编程的,译者推荐 Python 或者 Perl,或者其他功能强大的脚本语言.相对于较底层的 C/C++来说,这两种脚本语言都既有足够能力完成工作,又不会让你为底层操太多心.

请不必在"比较编程语言"这个环节上浪费太多时间.不考虑实际使用背景就比较语言毫无意义.在破解工作中能用得顺手,时刻让自己保持清醒的头脑才是最重要的.速度之类的重要性并不优先.

汇编的话,并不一定需要掌握.只是,掌握多少知识在破解上就能做到多少程度.许多时候并不

需要做到很深的程度就能完成破解任务了...至少,一部分吧

学习程序设计语言时,一开始要注意的是一些基本概念,如执行流的控制等;又例如面向对象语言里的"对象"等.只要在一门语言上清楚掌握了这些概念,就会发现在其他语言里其实都是一样的,只是具体写法可能不同.

过程化概念:

变量,以及变量的声明

语句,以及语句块

表达式与操作符

执行流控制,例如循环,分支,跳转等

函数的声明和调用

函数的重载

面向对象概念:

对象的概念

成员变量/方法与类变量/方法的区别

类的继承

...

基本上就是这么些概念了.

推荐书籍:

C++: C++ Primer, Effective C++

Java: Core Java 7th Edition

C#: C# Bible

Perl: Programming Perl(编程珠玑)

Python: 完了我不记得 Python 我看的是什么是...=_=

)

最重要的是,你得有解决问题的意识.你将着手的任务是谜题,而且它们并不会像你的计算机科学教授特意设计的题目一样,能正好用上这个星期的课教的内容以一页纸或不到的代码就解决.正好相反,你在与原作的日本程序员作战,而他们或许根本不在乎他们数据格式是否会被逆向工程所破解——如果他们在乎的话,他们大概更是会特地把数据加密来让你的工作更

难进行吧!

好了,卷起你的袖子,启动你的编译器,戴上你的"思维帽".在这个系列的文章中,我们会把一些范例游戏的攻破过程完整走一遍.希望你能看出自己是否适合破解者的角色...同时也学会几招道中小技巧吧!

(译者按: Part I 结束.上面提到了从游戏中提取图像资源的问题.嗯,也不排除有就是为了 CG 或者 BGM 而想加入到破解行列的读者.如果目的只是到这一步而已,那系列的后续文章基本上不看也可以,只要掌握一个基础工具就能够应付非常多的游戏了.

这个工具就是 **Susie**.

Susie 简介:

Susie 是一款在 Windows 上用于图像浏览的免费软件.凭借其良好的可扩展性,它不但可以用于浏览图片,在加上相应插件后也可以用于归档文件的资源提取,或者文件格式转换之类,因此在日本十分流行.所谓"有其他破解者写好了提取工具"很多时候就是以 Susie 插件(后缀为 spi, spi=Susie plug-in)形式发布的.

官网:「Susie の部屋」

<http://www.digitalpad.co.jp/~takechin/>

在此可以找到 Susie 的本体.

澄空学园 CK-GAL 区里的工具交流帖里有 Susie 本体以及相关插件包可以下载:

<http://bbs.sumisora.org/read.php?tid=207722>

此外如果需要给 Susie 寻找新插件,直接在搜索引擎上输入你希望提取资源的游戏名,加上"Susie""plug-in"等关键字就可以了.非常的多...

Susie 简单使用说明:

首先,Susie 是绿色软件,安装简单.使用归档管理软件(WinRAR,WinZIP,7-zip 等都可以)将 susie347b.lzh 解压到任意目录即可.

本体解压完成后,将插件也解压到 Susie 的安装目录下.Susie 在不安装任何插件的时候能做的事情并不多,所以在试图提取什么资源前务必确认是否已经正确安装好相应的插件.

要运行,如果不是在 J-Windows 或者当前语言区域在日语下的话,请使用 [AppLocale](#) 来引导 **Susie** 以日语启动;否则可以直接启动 **Susie.exe**.

启动后会见到一个工具条:

点击「開」,会看到下面的文件选择界面:

选择需要打开的文件即可.如果打开的文件是归档文件,那么可以直接从打开后的文件窗口里把需要的文件拖放到目标目录.

上面的文件选择界面里有"Catalog"选项,点击则会进入浏览/预览模式.

简单的拆档使用就这么简单.其他一些设置请查看 [susie.chm](#) 吧~

如果不介意使用付费工具的话,其实 [WESTSIDE](#) 对大量游戏都制作了资源提取器.当然其中不少也是 **Susie** 插件.

详细请见其官网, <http://www.westside.co.jp/index.html>

这个汉化技术区里的工具帖里还有不少其他工具,也很值得一看.)

Part II: 十六进制编辑器

(译者按: Part II 介绍的有两部分,其一是资源文件的常见形式,另一部分是在探究一个未知的资源格式时最可靠的伙伴——十六进制编辑器.在这部分的翻译开始前,译者希望能为缺少编程与计算机相关知识的读者先做些背景资料介绍.同时,在翻译过程中也会适当加入一些内容.

译者先前也写了点资源格式相关的文字,灯穗奇譚的文件格式与以 [ef - the first tale. / Trial Version](#) 中资源文件的加密的解析简单介绍几个工具中的[准备工作 1]部分)

(译者按: 背景知识介绍:

首先,数字的进制问题.

所谓"进制问题",小学生也应该知道"逢 X 进一"的就是 X 进制,所以也不需要多说...这里特地提到,是因为计算机所使用的进制与我们日常生活中习惯的十进制不同.这里要说的,是二进制和十六进制.

二进制,逢二进一,每位可以是 0-1 两个状态.

十六进制,逢十六进一,每位可以是 0-9,A-F 共十六个状态.下面十六进制数字都使用 0x 为前缀表示.没有 0x 前缀的则是普通的十进制数字.

不同进制间的换算就不用说了吧...不过 2 的补码(2's complement)需要说说.

计算机里常使用所谓"2 的补码"的形式来表达带符号的二进制数字.比起单独消耗掉一位来记录正负使 0 有两个,2 的补码完整的利用了整个 n 位二进制数 $[-2^{(n-1)}, 2^{(n-1)}-1]$ 范围内的所有数字.其值的计算方式是: $-2^{(n-1)} \times \text{最高位} + 2^{(n-2)} \times \text{次高位} + \dots + 2^0 \times \text{最低位}$.

非负的 2 的补码的二进制数与直接从十进制换算到二进制的一样,不过首位必须要为 0.

负的 2 的补码的二进制数是通过"取补"(complement)操作完成的.先得到数字的绝对值的原码,也就是直接从十进制换算到二进制的数;然后对每一位都"取反"(NOT),0 变为 1,1 变为 0,这样就得到了"反码";将反码加 1,得到"补码".

对 2 的补码形式表达的二进制数,加法直接进行;减法转换为加上减数的补码;补码的计算就是"取反加一",其作用就是取得原数的相反数(绝对值相等,符号相反).

当前主流的计算机使用的电子元件都只能支持两个状态间的切换,"开"或者"关".因而使用二进制数字来描述计算机上储存的数据非常合适.但是二进制数字写起来还是显示起来都很冗长,为了方便起见,使用 2 的 4 次幂=16 为底的十六进制来作为二进制数据的简写表达形式.所以,在查看二进制文件时,一般使用的是十六进制编辑器.高级语言像是 C++ 和 Java 里一般也只允许以八进制或者十六进制作为简写形式来表达二进制数据.

用十六进制来表达二进制数字的方法:

将原二进制数字以 4 位为一组,分别换算为十六进制的对应数字作为一个十六进制位.最靠左的一组二进制数字不足四位的在前面补零.这样,一个字节是 8 个二进制位,用十六进制表达就是 2 个十六进制位.

然后,"文件"相关.

在计算机上,一个"文件"是在次级存储器(如磁盘,磁片,磁带,光盘等)上储存数据所使用的基本结构.在面向对象程序设计里,文件是一种用于描述次级存储器与内存之间数据传输的软件对象.本段的讨论范围并不涉及"管道文件""设备文件"之类的特殊文件,请注意区别.普通的文件,可以分为文本文件和二进制文件两种.其中,

文本文件是能被人直接阅读的文件.一个文本文件里的数据由一串字符所组成.字符都是按照一定的文字编码所表示的,例如 ASCII 或者 UNICODE.可以直接以文本编辑器打开并阅读/编辑.举例的话,一个整数 12345,保存到文本文件之后,就变成了"1""2""3""4""5"这五个字符.

二进制文件是文本文件以外的文件.这种文件更加简洁高效,但是其数据通常是不满足文字编码的一串 0 和 1,不能直接被人阅读.举例的话,同样是整数 12345,保存到二进制文件之后,

就变成了 0x00 0x00 0x30 0x39 这 4 个字节的二进制数据.

Archive,意思是存档,档案文件.本文内将其称为"归档".这个名词应该并不陌生,因为电脑的日常生活中也经常会用到 rar,zip,tar,tar.gz,7z 等格式的归档文件.游戏多数都不会将其使用的资源直接分散放在游戏的安装目录下,而会将他们"归档"到归档文件里.这样就能够使文件结构更加清晰,而且也可以在一定程度上节省空间,一是归档的时候可能有压缩,二是大量小文件浪费空间的问题被解决了.因而,通常情况下,要提取资源,首要破解目标就是游戏所采用归档的格式.

游戏有可能使用同样的归档文件格式对其所有类型的资源进行归档,也可能对不同类型的资源采取不同的归档格式,也有可能选择性的归档.即使放入了归档文件,其中的资源也有可能被有选择性的压缩或加密过.

另外,还得提一下字节顺序的问题.

当一种数据需要多于一个字节来表达时,就牵涉到字节的顺序问题.可以认为从前向后读,首先读到的是最高位的,称为 big-endian 序;也可以认为最后读到的是最高位的,称为 little-endian 序.例如说,如果要读入一个 32 位的整型数,读入的数据是 0x61 0x62 0x63 0x64 的话,按 big-endian 读是 0x61626364,而按 little-endian 读是 0x64636261.同样的数据如果解释为 ASCII 编码的字符的话,无论采用什么字节顺序读都是"abcd"(也就是 0x61 0x62 0x63 0x64 对应的 ASCII 字符),因为 ASCII 字符只占一个字节(8 位),而字节序只影响同一数据内的字节排列顺序而不影响数据间的顺序或者字节内位的顺序.

在 Mac 的 PowerPC 上,数据一般以 big-endian 顺序储存.而在我们常用的 x86 兼容的 PC 机上,数据一般以 little-endian 顺序储存.请留意.

呵呵,正篇开始前似乎废话太多呢.好了,开始翻译了.)

那么让我们开始吧.今天我们会看看一个简单的范例归档格式,作为讨论一个"标准的"游戏数据文件的各部分的跳板.把这种模板记在心里,尝试理解游戏数据文件里看似无规律的字节时就会轻松不少.

这次我们作为范例的游戏是 Cross+Channel(体验版的下载链接在该页末尾).这个游戏已经有一个翻译计划正在进行中.喜欢的话下载一份体验版,然后我们来看看.

(译者按: 嗯,原文里提到的翻译计划自然是英文文化的翻译计划.现在其实也有中文化计划正在进行中.到时候或许就能见到成果了吧 ^^)

游戏安装后,安装目录下除了可执行文件之类以外,我们还能看到几个文件: bgm.pd, cg.pd, script.pd, se.pd, 还有 voice.pd.这非常典型: 多数游戏不会让每个音频文件和图像文件独立放在外面,而会把它们集合起来放在几个归档文件里,然后游戏引擎就可以随机访问它们了.把原始的独立文件从归档文件中分离提取出来是攻破游戏的第一步.

现在就该启动破解者最喜欢的工具,十六进制编辑器.这是一类相对简单的工具软件,只是显示文件的原始字节及对应地址...常见的高级功能包括将数据解释为常见数据形式(整数或者浮点数,之类),比较文件,还有搜索特定数据模式等.我个人习惯用 Mac 上的工具,HexEdit.如果你有喜欢的 Windows 或者 Linux 上的十六进制编辑器的话,可以在回复的时候提及,让其他读者也留意一下.

(更新: 至今提及的推荐的工具,在 Windows 上有 WinHex,HView,XVI32,Hex Workshop,和文本/十六进制混合编辑器 UltraEdit-32.Unix 系的操作系统上有 HexCurse.谢谢!)

(译者按: 十六进制编辑器最基本的显示部分有两个: 一是以十六进制方式显示的原始的字节数据,每一个字节表示为两个十六进制位,并且每个字节之间会稍微分开一点;二是于前者相对应的以 ASCII(或其他编码)方式解释的数据.有了对应的这个解释部分,我们就可以轻松的看出是否存在有明文存在的文本/标记.

译者想重点介绍一下 WinHex.它是不但可以打开一般文件,还可以像打开文件一样打开正在运行中的进程的内存,同时还有别的一些方便的工具,像是计算器,十进制与十六进制转换器,磁盘编辑器等...

例如说,在 Data Interpreter 窗口里,可以直接读出光标当前位置之后(包括当前位置)的 8 位,16 位和 32 位数据(以 little-endian 字节序解释)的十进制值,非常方便.)

那这些后缀为 pd 的文件看起来是什么样的呢? 下图是 cg.pd 的开头部分,我们可以肯定的猜测这个文件存有游戏的图像文件.

好极了,来看看: 可以辨认的文件名! (如 bgcc0000e.png) 如果你看到的是类似这样的东西,而不是一些看似随机的字节,你应该庆幸.虽然你可能还不知道这些数据到底代表着什么,很明显能看到继续解释这些数据的前路.

为了更好的解释我们看到的 data 是什么,是时候来简单介绍一下典型的游戏归档的内容了.

- 1. 文件头

文件头里包含的是关于整个文件的一般信息.并不是必须的.不过如果在一个归档文件的开头就看到一串不像是文件名的字符串的话,多半就是有文件头的了.

-- 1. 特征标记(signature)

通常一个归档文件都会以某种特征标记字符串开头,好让程序能辨认出归档的格式和版本.你可以通过这个标记来确认当前处理的文件是否属于正确的类型.

-- 2. 索引位置

大多数情况下,紧接着特征标记就会是归档内的内容索引了.不过有时候索引实际上位于归档的末尾,毕竟归档打包程序要等到归档内的内容都处理完了才会知道索引有多大(例如说

内容索引本身就被压缩过的情况).如果是那样的话,会有一个指向索引所在位置的指针.

- 2.内容索引

归档内容的索引是你需要掌握的重要结构,因为你要通过它才能知道如何提取出归档里的文件.

-- 1. 索引大小

索引一般会以一个表示大小的值开始.很多时候这个值就是归档所包含的文件数量,也可能是索引所占的字节数.不过索引大小并不一定存在,因为有时候内容索引会一直延续直到遇到一个特殊的结束记录(例如说,包含负的文件大小或者空的文件名的记录,等等).

-- 2. 文件记录的列表

接下来会是归档内所包含的每个独立文件的记录的列表.这可以是定长或者变长的数据结构,取决于文件名是如何处理的.有时候还会有表示目录树装结构的路径层次结构.每个记录含有含有一定数量的标准信息:

1. 文件名/文件路径: 可能是以 **0x00** 结尾的字符串,若不是也可能明确给出了字符串的长度.信不信由你,文件名其实不是必需的.我至少遇到过一个例子,只保存了文件名的**哈希值**.

2. 起始位置: 这会是一个相对某个位置的偏移量,通常是一个 **32** 位的整数.这"某个位置"可以是归档文件的开始,可以是内容索引的开始,或者有时候是"文件区"的开始(就是说,归档内第一个文件的起始地址,也可以说是内容索引的结束之后).

3. 文件大小: 文件的在归档内所占的空间大小,或者是文件的原始大小,通常是 **32** 位的整数.当文件在归档内所占空间与其原始大小相等时,文件大小要么有一个,要么干脆就省略掉了,因为可以从下一个文件的起始位置来计算出文件的大小;不相等时,通常说明文件被压缩过,则压缩后所占空间与原始大小都需要在内容索引里记录下来.

4. 标志位: 标明文件是否被压缩过,有的话用了何种算法;或者是否被加密过,有的话是否有相应的密钥或者初始值等.

5. 校验和(**checksum**): 为确保数据的完整性,有时候会记录下文件的校验和.这对破解者来说可能有点烦,因为这意味着修改归档的内容后我们还得把校验算法也跟出来,才能计算出修改过的新数据的正确校验和(不然想办法禁止掉可执行文件里的校验检查也可以).

注意: 有时候这些信息可能会分散在不同位置.例如说,文件的起始位置与文件名放在了索引的记录里,而文件大小和相关的一些标志位却在那个起始位置给出,紧接着的就是相应的原始文件.

- 3.原始文件

原始文件的数据基本上就是头尾相接的放置在归档文件里了.这些数据有可能被压缩过也有可能被加密过.游戏引擎可以通过索引快速的定位到这些数据,因而可以任意使用它需要的文件数据.

好吧,了解了这个标准模板后,让我们来看看它能如何解释 **cg.pd** 里的数据.最开始的 **PackOnly** 看起来像是个特征标记.接下来是一串 **0x00** 字节,直到我们来到地址 **0x40**,在一串可辨认的 **ASCII** 字符串之前有这么一组数据: **0x21 0x02 0x00 0x00 0x00 0x00 0x00 0x00**.

这会是内容索引的大小吗? 说起来,我们应该如何解释这几个字节呢? 我们有几种选择:

·**随机的标志位**. 这里可以看成 3 个位被置位(set)了: 第一个字节中的 0x20 和 0x01,以及第二个字节中的 0x02.(注意这里最好转换到二进制观察每一位的值,是置位(set)还是清零(clear).例如 $0x21 = 00100001 = 0x20 \text{ AND } 0x01$.)这么解释算是有点道理,不过暂时没什么价值.看看其他可能吧.

·**little-endian 整数**. 这里,0x21 是最低字节,0x02 是次低字节,依此类推.所以这个整数的实际数值是 0x0000000000000221,十进制就是 545.这可能是一个合理的索引大小值.

·**big-endian 整数**. 按这种字节序的话,0x21 是最高字节,0x02 是次高字节,依此类推,整数值是 0x21020000,也就是十进制的 553,779,200(这已经是忽略了后面那 4 个 0x00 了).这个数据比较不合理,因为整个归档文件才不到 500MB.所以如果想以 big-endian 方式来解释的话,得换个长度,例如说这可能是个 16 位的整数: $0x2102 = 8450$,这个值或许有可能,例如说是索引数组的字节数之类.

让我们来具体看看.从归档文件的开头开始看下来,似乎文件名是在 0x48,0xD8,0x168,0x1F8,0x288 等位置开始的.也就是说每个文件名的开始到下个文件名的开始之间有 $0x90 = 144$ 字节.最后一个文件名(TCYM0005c.png)是在 0x13248 开始的,说明大概有 $(0x13248 - 0x00048) / 0x90 + 1 = 545$ 个文件记录.

你看到了什么了么? 没错,就是 545! 以 little-endian 方式来解释 0x21 0x02 0x00 0x00 0x00 0x00 0x00 0x00 应该正确的告诉了我们内容索引里有多少条记录.而且更重要的是,现在我们可以专注于那些 144 字节的数据,确信这些就是一个一个的索引记录.而且,我们知道这个归档青睐 little-endian 字节序,也可能使用 8 字节(64 位)的整数.

那就让我们来看看这些索引记录.不过(这里有个小技巧了)不要看第一个记录.很多数据在第一个记录里都有可能是零,我们就看不出数据的意义了.来看看第二条记录吧,在地址 0xD8 到 0x167:

我们看到了一个文件名,一堆零,和看起来像是 8 字节的 little-endian 整数.回忆一下典型游戏归档的模板...我们在寻找的是文件大小和位置的信息,或许还有一些标志位之类.那堆 0x00 可能会是标志位,不过现在还没办法确定.

眼下先假设那些零是文件名所属的数据结构的一部分: 这样就刚好给文件名分配了 128 个字节,是一个(懒惰的?)程序员会做的合理的事情.剩下的信息是两个整数,0x002420FA 和 0x0008370E.暂时还不知道这些是什么数据...还是先多看几个记录吧.内容索引的头几条记录里对应位置的数据是什么样的呢?

File 1 0x00240048 0x000020B2

File 2 0x002420FA 0x0008370E

File 3 0x002C5808 0x00002FA6

File 4 0x002C870E 0x00063B8A

File 5 0x0032C338 0x0006A7CB

现在这些数据有点看头了.第一列数据总是越来越大,更重要的是它们总是以第二列的值增大! 这正是经典的位置+文件大小的进行.

如果我们的假设是正确的,那么第一个文件,bgcc0000e.png,应该有 $0x000020B2 = 8370$ 字节这么长,而且应该在归档内的地址 0x00240048 附近开始.我们不能完全肯定,因为不知道这个偏移量是相对归档文件开头还是特征标记后还是哪里,而且这个值有点诡异,因为我们知道内容索引是在0x000132D7结束的,还记得吗? 总之先到那个地址去看看吧,因为文件的顺序可能被打乱了

爽! 我们猜得完全正确,地址 0x00240048 正是一个 PNG 文件的开头.无压缩,无加密.事实上,要是我们从这个地址开始把接下来的 8370 字节复制粘贴到一个新文件里(当然也是在十六进制编辑器里),然后用图像浏览器打开的话,我们就得到了...一张 640×480 的空白白色图.

呃.无论如何,游戏也是需要纯白图的,至少这图的尺寸没错.那么为保险起见,再试试下一张图片.从地址 0x002420FA 开始复制出 0x0008370E 个字节,我们得到的是:

好耶! 胜利是属于我们的!

现在让我们来总结一下.我们在这个阶段,认为.PD 格式包括:

特征标记字符串"PackOnly"

56 字节的 0x00

8 字节 little-endian 的文件数量值

多个 144 字节的索引记录,每个包括:

128 字节的文件名,是以 0x00 表示结束的字符串

8 字节 little-endian 的文件位置(从归档文件开头算起的偏移量)

8 字节 little-endian 的文件大小

最后是原始文件数据,无压缩无加密,正好在索引里给出的位置上开始.

那么,内容索引之后到第一个原始文件之间的这段空白(全是 0x00)该如何解释呢?

其实,那个地址,0x00240048 看起来很可疑...一个索引记录是在 0x48,也就是说有 0x240000 字节的空間可用于放置索引记录.每个记录 144 字节的话,就能装下 16384 个记录.也就是 2^{14} .所以让我觉得这很像是一个(懒惰的?)程序员会做的事: 留下足够多的空间给大量的文件用就算了.

那这段空白是否必要呢? 说不定我们在重新打包归档的时候可以把这段空白清除掉,省下那

么几兆空间.要不然我们把数据移动超过 1 个字节程序也会崩溃...我们只能等后面实践的时候才知道了.

好了,下次就让我们把获取到的知识转换成实际的程序代码吧.当然,我们会遇到些障碍,嘿嘿

(译者按: Part II 结束.这个 part 所讲解的例子是一个非常简单,无加密无压缩的归档文件的格式分析.

简单说来,如果被分析的归档文件比较典型且无加密无压缩,那么就按照典型的游戏归档的形式,找到内容索引后,猜测索引内每个数值的意义,并且到归档内猜测的位置寻找原始文件.

译者的经验是,如果一开始就以脚本文件为目标的话,过程会比较痛苦.因为脚本文件经常是纯文本文件或者一些特制的格式,不一定有明确的起始标示,不便于确定是否正确定位到了原始文件的位置.所以,可以尝试对估计含有图像的或者音频的归档文件下手,就像本篇的例子以 CG 归档为破解对象.

为什么要针对图象,音频和视频下手呢? 因为业界在许多时候都会使用标准的格式来储存图像,音频和视频文件.

下面列举几种常见的文件格式,以[格式名]: [特征标识串]表示

图像:

BMP: 0x41 0x4D (BM)

PNG: 0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A (.PNG....)

GIF: 0x47 0x49 0x46 0x38 0x37 0x61 (GIF87a) 或 0x47 0x49 0x46 0x38 0x39 0x61 (GIF89a)

**JPEG: 0xFF 0xD8 0xFF 0xE0 0xxx 0xxx 0x4A 0x46 (. ...JF)
0x49 0x46 0x00 (IF.)**

音频:

OGG: 0x4F 0x67 0x67 0x53 ("OggS")

**WAV: 0x52 0x49 0x46 0x46 0xxx 0xxx 0xxx 0xxx (RIFF....)
0x57 0x41 0x56 0x45 0x66 0x6D 0x74 0x20 (WAVEfmt)**

视频:

MPEG: 0x00 0x00 0x01 0xBx

**AVI: 0x52 0x49 0x46 0x46 0xxx 0xxx 0xxx 0xxx (RIFF....)
0x41 0x56 0x49 0x20 0x4C 0x49 0x53 0x54 (AVI LIST)**

在找到这些特征标识串后,我们就能轻松确定 1)是否存在某类型文件 2)文件的起始位置从而可以与文件头里的信息进行对比,判断数据的意义,然后推广到游戏中同格式的其他归档的处理(例如含有脚本的归档)

更多更详细的文件特征标识串,可以在这里查询:

http://www.garykessler.net/library/file_sigs.html

Part II 里举的例子"太过典型",让我们来看看可能发生什么简单的变化吧.同样是无加密无压缩的归档,灯穗奇譚里的 AOD 格式的归档虽然也有内容索引,但却被分成了一段段.详细请看灯穗奇譚的文件格式.

另外,区里另一篇帖子的例子更加有趣.[ONE ～輝く季節へ FullVoice 汉化实战篇](#),其中的归档与其索引是分开在不同文件里的.再次提醒我们索引的必要性以及可能需要变通的地方.

不得不注意到,并不是所有游戏都会把资源都放入归档内的.

举个例子来说,Visual Art's 旗下制作组所使用的 RealLive,会使用 GAMEEXE.INI 文件来配置是否使用归档.其中相关的一段:

Quote:

```
#FOLDNAME.TXT = "DAT" = 1 : "SEEN.TXT"
#FOLDNAME.DAT = "DAT" = 0 : "DAT.PAK"
#FOLDNAME.ANM = "ANM" = 0 : "ANM.PAK"
#FOLDNAME.ARD = "ARD" = 0 : "ARD.PAK"
#FOLDNAME.HIK = "HIK" = 0 : "HIK.PAK"
#FOLDNAME.PDT = "PDT" = 0 : "PDT.PAK"
#FOLDNAME.G00 = "G00" = 0 : "G00.PAK"
#FOLDNAME.M00 = "M00" = 0 : "M00.PAK"
#FOLDNAME.WAV = "WAV" = 0 : "WAV.PAK"
#FOLDNAME.BGM = "BGM" = 0 : "BGM.PAK"
#FOLDNAME.KOE = "KOE" = 1 : ""
#FOLDNAME.MOV = "MOV" = 0 : "MOV.PAK"
#FOLDNAME.GAN = "GAN" = 0 : "GAN.PAK"
```

中间的数字就是说明是否使用归档的,是的话值为 1,否的话值为 0.这个还要与后面的文件名相配合,即使前面的值为 1,假如后面的文件名为空串的话,也不使用归档,而是直接把一个个资源文件独立放置在安装目录下.上面这段引用自智代 After 的 GAMEEXE.INI,可以看到只有脚本资源被放进了归档里,文件是 SEEN.TXT.把这个归档文件拆开,就能看到里面实际上是许多小文件,SEEN0628.TXT 到 SEEN9072.TXT.而这些小文件又是编译过的脚本.

就算是放进了归档里的资源,要分析其内容索引也不总是这么轻松.假如说有数据被加密或压缩过,情况就会相对复杂一些.

也举个简单的例子吧.呵呵这个可是运气/RP 大爆发的例子...

在はるのあしおとの web_trial 里,归档文件的后缀是 paz,都被简单加密过.要使用上面的经验来处理加密过的归档显然不实际,至少也得先解决解密问题.怎么办呢?

由于业界经常在音频格式上选择使用 Ogg Vorbis,而且在游戏的安装目录下发现了"ogg.dll"和"vorbis.dll"这两个用于处理 Ogg Vorbis 文件的程序.所以我们大胆猜测 bgm.paz 里包含的背景音乐文件是采用 Ogg Vorbis 格式的,并由其入手.

用十六进制编辑器打开 bgm.paz,发现里面都是些无法识别的数据.从头到尾浏览过之后没有发现形似文件头或者内容索引的东西.因此猜测文件被加密过.

上面说明文件的特征标识串时提到了,Ogg 格式的文件是以 0x4F 0x67 0x67 0x53 ("OggS")开头的,注意到中间有连续两个字节的值是一样的.如果运气好,文件只是被简单的加密过的话,那么加密后的密文里这两个字节的对应位置的值也应该是一样的;同理,拥有相同值的连续两个字节,其之前和之后的字节的值应该不同.根据这条线索,让我们来找找文件中有符合这个特征的地方.

很快我们就注意到了这个地方.红色标记出来的部分就是我们找到的一组符合线索的数据,0xB1 0x99 0x99 0xAD.先看看 0x67 可以如何对应到 0x99 上.还记得本篇开始时背景知识里提到的 2 的补码吗?把 0x67 和 0x99 分别展开,可以看到它们正是符合互为补码的关系.将头尾两个字节也对比一下,发现符合相同的关系.于是可以大胆猜测,整个文件都是以取补码的方式简单加密过的(虽然这个猜测不完全对...嘿嘿不过还是很 RP 吧...).

于是把想法实践一下,发现简单解密处理处理过后的文件已经相当的可读了.这样就可以继续按照典型的归档文件的处理方法来处理了.

注意到,这里 bgm.paz 的头 4 个字节我并没有做取补处理.原因见[以 ef - the first tale. / Trial Version 中资源文件的加密的解析简单介绍几个工具中的\[准备工作 1\]](#)部分所写的内容,这里就不重复写了.

虽然通过简单的观察就能破解出归档格式是很 RP 的事,不过游戏厂商在决定使用什么加密/压缩方式时本来也很 RP -- XD

简单观察法适用于无加密无压缩,或者只做了简单加密的归档.简单的加密方法有: 1)加上或减去一个常量; 2)取反(NOT)或者取补(complement/negate); 3)与常量做简单的异或(XOR).如果是上述的 3 种情况,那么加密后的密文里字节与字节间的相等/不等关系会得以维持,因而可以让简单观察法顺利进行.

再举个例子的话...嗯,CIRCUS 虽然一直不怎么用归档,不过它的游戏的脚本文件(后缀 MES)对文本部分加了密,用的就是加上常量的方式,所以很容易由观察得出.

Part III: 构建代码原型

(译者按: Part III 可以说是"实战"的重要部分,也回答了"高级程序设计语言在游戏破解中有什么用"的问题.在这篇里,Ed 介绍了以 Python 语言编程为手段,对 Cross+Channel 的归档文件进行拆包,打包,修改等为翻译而做的破解中必须经过的一些步骤.在开始翻译正篇前,照例也写点背景资料吧.不过这篇原文里的经验成分就非常丰富了,相对来说译者也没有多少可发挥的空间.毕竟译者也缺乏经验,才入门没多久...)

(译者按: 背景知识介绍:

在开始编程之前,我们需要知道在这篇的范围内,编写程序的意义.这是个相当浅显的事实:程序能做到的,手动也肯定一样能做到.编写程序的意义就是为了提高过程的自动化程序,让电脑做它最擅长的事——完成烦琐的计算和重复的操作.除非...你有毅力,能手动把成百上千,甚至上万的小文件慢慢复制粘贴出来,修改后再复制粘贴出一个新文件;又或是遇到简单的加密时用原始的纸笔计算来完成整个解密过程...不然,还是写点程序吧.一劳永逸.虽然不否认有人能做到,复杂的加密和无论复杂与否的压缩,徒手完成所有计算不是一个普通人能在合理的时间内完成得了的事 ^ ^

从某种意义上说,拆包程序的核心就是把我们对归档文件的结构的了解转换为一个循环:
(在归档级做解密或解压)-读取内容索引信息-读取原始文件信息-(在原始文件级做解密或解压)-写出处理过的原始文件数据.

如果归档里索引位于文件的开头的话,那么打包程序的核心就会是这样的一个循环:
(在原始文件级做加密或压缩)-收集索引所需信息-依次写入内容索引-依次写入处理过后各原始文件(-在归档级做加密,压缩或计算校验和).
相当的直观吧? 其实拆包/打包程序一点也不神秘,多数情况下都只需要会编写最基本的文件输入/输出程序就能完成.

学习编写拆包/解包的程序,你将能更充分的理解归档内各种数据(特别是索引内,每个记录里的数据)有什么用,如何使用等.从程序的需求的角度出发,一切都会变得明了.切记,归档内没有什么数据是真的没有任何作用的;如果你觉得有"无用"的数据,多数情况下说明你尚未理解该数据的使用方式,反而应该留心注意.

举例的话,Cross+Channel 的归档就可以.就像 Part II 里提到的,内容索引里记录了文件大小.但是我们知道,文件大小并不总是必要,因为可以从下一个文件的起始位置来推断出当前文件的大小 = 当前文件的起始位置 - 下一个文件的起始位置.编写 Cross+Channel 程序的程序员似乎傻乎乎的浪费了空间存了不需要的数据呢.真的是这样吗? 阅读下面的正篇你就会知道,Cross+Channel 的归档里文件的顺序并不固定;也就是支持乱序存放,索引记录先出现的对应文件不一定在索引记录后出现的对应文件之前.不记录下文件大小,游戏引擎(当然我们将要编写的处理程序也一样)要处理这种归档就会非常麻烦.

再举 minori 的是るのあしおと或 ef 之类的例子,归档里每个记录都有 3 个同文件大小相关的值,而且经常 3 个值或至少其中头 2 个值是相等的.为了了解其意义,需要寻找 3 个值都不同的情况并加以分析.

回到编程的问题.嗯,我们说要编程,但是什么叫做构建代码原型(code prototyping)?
因为我们对归档格式的猜测很有可能不完善甚至有根本上的缺陷,一开始编写出来的程序代码有可能经过多次不同程度的修改.如果一开始就想把代码向"最终发行版"的方向写,一味注

重程序的运行速度,再写上一堆图形界面相关的内容弄得花里胡哨,注意力就会分散,且被频繁修改的代码也有可能很乱.所以,我们可以构建代码原型,先把最重要最核心的部分写出来并加以测试,在反复修改确认其正确性后,如果有兴致可以方便顺利的写出所谓的"最终发行版".如果在程序设计语言上有个人偏好,甚至可以在构建原型与后期编写发布用程序时使用不同的语言;有些语言做脏活累活确实是比较方便...

Part III 的正篇里,Ed 的一个重要技巧非常值得提倡,那就是不要只顾着以"已知"的理解方式来处理归档,而要记得检验"不确定"的部分是否也与猜测的相同.因为我们在猜测一个归档的结构的时候,多数情况下都不会手动把归档内所有文件都解出来来验证猜测的正确性,所以很有可能漏掉了一些数据或者一些例外.写好原型代码后,我们就能快速的定位到这些例外所在,然后回到十六进制编辑器里继续分析.

对程序设计语言不熟悉的,请尽快对各种运算符熟悉起来.这样至少读代码的时候能知道在发生什么事.

许多语言里的运算符都与 C 的相同或相似,所以以 C 为例的话,基本有:

算术运算符: + (二元加), - (二元减), - (一元取相反数), * (乘), / (除), % (取模,也就是相除后的余数)

逻辑运算符: && (与, AND), || (或, OR), ! (非, NOT)

按位运算符: & (按位与, bit-wise AND), | (按位或, bit-wise OR), ~ (按位非, bit-wise NOT), ^ (按位异或, bit-wise XOR)

移位运算符: << (左移位), >> (右移位)

注意到单个等号"="是赋值符,在上面除&&和||外其他运算符后接上一个等号是简写的运算并赋值.要验证相等性的话,用两个等号"==",切记.

在 C-like 语言里把"="与"=="是常见手误之一,要小心.

C-like 语言中如 Java,C#,JavaScript 等会有>>>的按位操作符,意义是无符号右移位.

如果对逻辑运算不熟悉的话,下面列出了 4 中最常见逻辑运算的真值表,T表示真(true),F表示假(false):

与 AND: 二元运算.只当所有操作数都为真时,结果为真

AND T F

T T F

F F F

或 OR: 二元运算.只要操作数中有为真的值,结果就为真

OR T F

T T T

F T F

非 NOT: 一元运算.将操作数的真假值反转

NOT T F

F T

异或 XOR, exclusive-or: 二元运算.只有当操作数不同真假时,结果为真

XOR T F

T F T

F T F

由于计算机使用的二进制数字,每一位只能有0和1两种状态,与逻辑运算里的真假正好吻合,所以逻辑运算也用于位对位的二进制数运算上,也就是按位运算.这时,0 等价于 `false`,1 等价于 `true`.

又说多了...赶紧进入正篇 ^ ^)

在上一部分里,我们用可靠的十六进制编辑器分析了 `Cross+Channel` 的归档文件格式.在概念验证(`proof-of-concept`)性质的手工抽取图片后,我们相信我们已经知道那个归档文件是怎么回事.但是,要想确定的话只有一个办法:写一些工具,然后修改游戏试试看!

今天我们会尝试快速的代码原型构建,编写一些适用于处理那些归档文件的工具.我的意图是让这过程尽可能的简单直观,这样,如果你看完这部分之后会说:"你是说要破解一个游戏只要写那么点代码吗?!" 那么...任务完成.

我们应该使用什么语言呢? 平时我会用 `C`,不过我也想从这系列的文章中获取点什么,所以我们转而使用 `Python`,对我来说也是一种学习经验.这种语言的主要优势在于:

- 跨平台. 它不但在 `Windows`, `Linux`, `Mac OS` 上都可用,而且还统一了一些操作系统特有的特性,如目录分隔符.

- 适用于广泛的范围. 它提供了许多高级数据结构,如哈希字典(`hash-table dictionary`)和 `function continuation`,同时也提供适当的低层的位操作等.

(译者按: 由于译者见识尚牵,没在中文资料上见过 `function continuation`,翻不出来 `=_ =` 笼统来说就是可以将当前运行状态保存下来留待以后使用,就像把栈压到了栈里一般.事实上 `Python` 还支持一种较少见的数据结构 `closure`(闭包),意味着可以嵌套函数调用并将其返回.)

- 直观的语法. `Python` 几乎如同可执行的伪代码一样,所以如果你知道任何其他语言,你都应该能轻松读懂 `Python` 代码.

- 交互模式(`interactive mode`). `Python` 解释器可以在交互模式下运行.如果你想的话,可以在交互模式下把指令一行一行输入,代码会得到立即执行.这对构建原型时避免重复的编辑-重

编译-运行测试的过程有相当的好处.

很不幸的是,Python 的主要缺点是运行速度慢.Python 在对付位操作时的速度确实不能与 C 或者其它彻底的编译语言相提并论.我们这里的应用场景下这倒还不算太糟糕,而且就算慢得值得注意的话,其实也可以通过良好的 Python/C 接口来搭配代码.

(译者按: 其实随便拿一种语言都能写出一堆优缺点.要说跨平台,只在运算和基本的文件输入输出上,C/C++,Java,Perl 等一样跨平台,虽然可能得重新编译...要说易懂的语法,其实 C-like 的语言长得都差不多一个样,光是"读懂"恐怕不会是什么问题.译者想再次提起的是,选定了一种语言就用就是了,考虑到我们这里的使用场景,对语言太挑剔是无谓的.

不过 Python 有个重要优点,Ed 似乎没提到.Python 是使用缩进来定义代码块的,因而写出来的代码几乎天生就有良好的风格.有些人或许对强制缩进表示反感,译者认为至少这对读代码的人是很大的好事.

说起来 haeleth 对 OCaml 很执着...无论到哪里他都不忘宣传一下 OCaml 的好处,呵呵,原文的回复里他还顺便踩了踩 Python...)

如果你使用 Mac OS X,你很可能已经有现成可用的 Python 了.如果使用 Linux,你可能已经装了,否则也可以轻松下载到一个安装包.如果使用的是 Windows,这里也有为你准备的安装程序.

我们需要开始编写一些什么样的例程(routine,在不同语言里也可能叫做函数(function)/过程(procedure)/方法(method))呢? 还是让我们先回顾一下上次确定的归档格式:

特征标记字符串"PackOnly"

56 字节的 0x00

8 字节 little-endian 的文件数量值

多个 144 字节的索引记录,每个包括:

128 字节的文件名,是以 0x00 表示结束的字符串

8 字节 little-endian 的文件位置(从归档文件开头算起的偏移量)

8 字节 little-endian 的文件大小

最后是原始文件数据,无压缩无加密,正好在索引里给出的位置上开始.

那么看起来我们需要下面的一些例程:

- 读/写特定长度和字节序的整数
- 读/写以 0x00 结尾的字符串
- 检查特征标记和已知的零串

轻松! 这类实用例程以后起来会很顺手,所以写了之后我们也可以在后续处理的游戏里使用.
作为例子,让我们先解决掉整数的处理:

```
def read_unsigned(infile, size = 4, endian = LITTLE_ENDIAN) :
    result = long(0)
    for i in xrange(size) :
        temp = long(ord(infile.read(1)))
        if endian == LITTLE_ENDIAN :
            result |= (temp << (8*i))
        elif endian == BIG_ENDIAN :
            result = (result << 8) | temp
    return result
```

(译者按: 如果你是刚接触 **Python** 的,请千万注意缩进.在论坛上贴出来的代码的缩进可能乱了,千万注意)

取决于你的经验,上面这段代码片段可能是微不足道的也可能让你完全摸不清方向.基本上,它只是从文件里一次读取一个字节,然后按照字节序的规定(最低字节在前(**little-endian**)或者最高字节在前(**big-endian**))把那些位移动到正确的数位上来构成一个多字节数字.

我们需要的另外几个例程也是类似的,说实话也没必要在这里详细讨论.我写了一个实用例程包,需要的话可以[下载 insani.py](#) 来用.

现在让我们来看看问题的关键部分,读写归档!

基本上我们只要遍历一遍上面我们的归档格式,并将其转换为处理代码就可以了.一开始是特征标记和文件头:

```
from insani import *
arcfile = open('cg.pd', 'rb')
assert_string(arcfile, 'PackOnly', ERROR_ABORT)
assert_zeroes(arcfile, 56, ERROR_WARNING)
numfiles = read_unsigned(arcfile, LONG_LENGTH)
print 'Extracting %d files...' % numfiles
```

这里我从 **insani.py** 里用了 **assert_string()**和 **assert_zeroes()**这两个实用例程来检查我们预期的特征标记和一串零.常数 **LONG_LENGTH** 就是 8,归档里使用的整数的长度.**Little-endian** 作为缺省值,实际上隐式表达了.

这里要注意的重要问题是错误检查.我不是在说"常规的"错误检查,例如确保文件已打开,或者是否有足够内存读入一个 8 字节的整数之类.那类东西对一个原型工具来说有点多余: 我们才不在乎文件不可读时程序会崩溃,事实上我们反而预期着这样的行为.

不,我所说的错误检查是指检查我们对归档的假设.你看,我们完全可以跳过头 64 个字节,直接跳到读 `numfiles` 的语句.其实那才是我们所需要的数据,不是吗?为什么我们要特地检查特征标记和那些零呢?

我们这么做是因为我们还在检验我们对这个归档文件的理解...我们其实只看了一个游戏里一个归档里的很小一部分.那 56 个零真的没有用吗?总是没用?我们无法确定.所以如果我们写的工具读到了它没有预期到的数据——就是说,我们没预期到的数据——那我们就得知道发生了什么事,要么在"我完全混乱了"的时候中止程序,要么在没那么关键的不符情况至少输出一条警告信息,这样我们可以观察问题并更仔细的改进工具.

好了,这个问题说得够多了.相信我: 多写检查你的假设的代码而不要直接跳到"已知重要"的地方. 现在让我们继续读取内容索引:

Copy code

?

```
print 'Reading archive index...'
entries = []
for i in xrange(numfiles) :
    filename = read_string(arcfile)
    assert_zeroes(arcfile, 128-len(filename)-1)
    position = read_unsigned(arcfile, LONG_LENGTH)
    size = read_unsigned(arcfile, LONG_LENGTH)
    entries.append( (filename, position, size) )
assert_zeroes(arcfile, 144*(16384-numfiles), ERROR_WARNING)
```

这里我们用了点 **Python** 的神奇之处来让问题更简单.在读取文件名,位置和大小之后,我们把这三个变量收集到一个"tuple"里,以多出的一对括号表示,并把它插入到 `entries` 链表里.在 **C** 里做同样的事情会稍微复杂一些,而在这里几乎只是伪代码一般.再次,请注意断言语句(**assertion statement**): 这个文件里的每一个字节都要得到检查,无论是有用的数据还是我们不感兴趣的地方.

现在,我们完成整个过程,实际写出文件,并且稍微做一下清理工作:

```
for (filename, position, size) in entries :
    print 'Extracting %s (%d bytes) from offset 0x%X' % \
          (filename, size, position)
    outfile=open(filename,'wb')
```

```
        assert (arcfile.tell() == position)
        outfile.write( arcfile.read(size) )
        outfile.close()
assert (arcfile.read(1) == '')
arcfile.close()
```

(译者按: 注意这里第二行末尾那个"\",这意味着将接着的换行符转义掉,于是可以把对写在一行上太长的代码写在多行上)

这里,循环的结构又是 **Python** 的一个好地方: 我们可以毫不费事的遍历 *entries* 链表,每次取出整个 **tuple** 并且再次使用那些变量.与其用 *assert* 语句,我们本来可以使用 *arcfile.seek(position)*指令来直接跳到我们想要的地址,但是我们还在检查我们对归档格式的了解: 归档里文件是否都如我们所料,按顺序放置并且中间没有间隙? 在处理完所有文件后,我们真的来到了归档的末尾吗,还是说还有被遗漏了的数据?

如果我们都做对了,那就不应该有任何意外发生.那就让我们来试试看吧.如果想的话,你可以下载 **crosschannel-extract1.py**,不过我建议你自己实现一次,至少自己敲入这些代码来找到这种编程语言的感觉:

```
% python crosschannel-extract1.py
Extracting 545 files...
Reading archive index...
Extracting bgcc0000e.png (8370 bytes) from offset 0x240048
Extracting bgcc0023.png (538382 bytes) from offset 0x2420FA
...
Extracting TCYM0005c.png (156881 bytes) from offset 0x566E1B4
```

看起来工作正常了,而且所有提取出来的 **PNG** 文件都可以用你最惯用的图像编辑器打开.唔,不过之中有一些是 **NSFW**,顺便一提 ^ ^; (**NSFW** = **Not Safe For Work**,意味着有糟糕内容 XD)

验证概念的提取工具成功了.现在我们可以稍微改进一点,因为硬编码在代码里的归档文件名肯定算不上是好事.改进过的版本是 **crosschannel-extract2.py**,里面增加了一些 **Python** 的操作系统接口库中的例程来从命令行读取参数,也可以那文件提取到新的子目录里,之类.我也稍微调整了一下错误检查的代码,允许乱序放置的文件,同时也在看到"明显"不正确的地址或文件大小时尽早中止程序.有兴趣的话可以看看,不过那对于继续推进这篇文章的进度并不重要.

我们现在确实应该做的事是重建归档! 没问题.我们可以把刚才的代码拿来,轻松的改换为对相反的操作上.首先我们需要收集要打包的文件的信息,从命令行传入的子目录名获得:

Copy code

[?](#)

```
import sys, os
from insani import *

dirname = sys.argv[1]
rawnames = os.listdir(dirname)

numfiles = 0
position = 144*16384+72
entries = []
for filename in rawnames :
    fullpath = os.path.join(dirname, filename)
    if os.path.isfile(fullpath) :    # Skip any subdirectories
        numfiles += 1
        size = os.stat(fullpath).st_size
        entries.append( (filename, fullpath, position, size) )
        position += size;
```

在收集信息的过程中,我们也可以基于文件的大小和已知的文件头大小来计算出每个文件的最终地址.然后我们就可以把它们写出来:

Copy code

[?](#)

```
print 'Packing %d files...' % numfiles
arcfile = open(sys.argv[2], 'wb')
arcfile.write('PackOnly')
write_zeroes(arcfile, 56)
write_unsigned(arcfile, numfiles, LONG_LENGTH)

print 'Writing archive index...'
for (filename, fullpath, position, size) in entries :
    write_string(arcfile, filename)
    write_zeroes(arcfile, 128-len(filename)-1)
    write_unsigned(arcfile, position, LONG_LENGTH)
    write_unsigned(arcfile, size, LONG_LENGTH)
write_zeroes(arcfile, 144*(16384-numfiles))
```

其实这相对简单些,因为我们不需要再检查什么了: 我们只需要写入我们知道的东西...没有任何不确定因素.你也会注意到这与我们开始所描述的归档结构的对应性: 有实用例程是件好事,这样我们就可以用一两行代码来解决结构里的每项内容.现在我们把要打包的文件本身也写入归档里:

```
for (filename, fullpath, position, size) in entries :
    print 'Packing %s (%d bytes) at offset 0x%X' % \
        (filename, size, position)
    assert (arcfile.tell() == position)
    infile = open(fullpath,'rb')
    arcfile.write(infile.read(size))
    infile.close()
arcfile.close()
```

这里我们用 **assert** 语句来做了点正确性检查,不过那其实只是个"愚蠢的程序员"的检查而不是别的: 如果我们写出的数据都是正确的,那么这个断言(**assertion**)永远都为真,无论归档格式的细节如何.这是个典型的"失败就中止"式的断言用法: 用来快速的检查如果一切操作都正确时不可能出错的东西.

如果喜欢,你可以下载 **crosschannel-repack1.py**,不过我还是鼓励你自己写一次.让我们来测试一下:

```
% python crosschannel-repack1.py temp temp.pd
Packing 545 files...
Writing archive index...
Packing bgcc0000a.png (408458 bytes) at offset 0x240048
Packing bgcc0000b.png (436171 bytes) at offset 0x2A3BD2
...
Packing xiconp.png (2399 bytes) at offset 0x5693D26
```

一个成功的工具的**金牌标准**就是,能够把一个归档解包并重新打包后,得到与原来的归档每一位都相等的复制品.我们成功了吗?

```
% diff cg.pd temp.pd
Binary files cg.pd and temp.pd differ
```

(译者按: 在 Windows 的命令行里,对应这里 *diff* 的命令是 *comp*,格式是 *comp 文件/目录名 1 文件/目录名 2*

呵呵,上面的信息意味着被比较的两个文件内容不相同)

呃,糟糕.不过,其实这也是预料中事...文件打包的顺序不一样,因为我们打包的时候文件是按字母顺序排列的,而原始的归档里则更为随机.不过这有关系吗?我们还不肯定.

一个成功的工具的**银牌标准**就是能够成功的把自己打包的归档解开.所以,我们从原始归档提取,重新打包,然后再尝试提取,以确认两次提取到的数据是否相同.

```
% python crosschannel-extract2.py temp.pd temp2
Extracting 545 files...
...
% diff -r temp temp2
```

成功.好吧,至少我们的解包和打包工具自身是兼容的.不过,它们跟游戏相兼容吗?

这里要介绍个道中小技巧.在第一次测试重新打包的归档时,什么都不要改变.把游戏自身的归档文件解出来,重新打包,然后看看游戏是否还能正常运行.这样你就能看出,例如说游戏在可执行文件里保存了归档文件的 **MD5** 校验和,又或者你对归档格式的理解有所遗漏之类.你解决了这些问题,然后再去修改文件并插入英语之类.

(译者按: 当然对于中文化工作者来说,修改文件插入的会是中文啦 ^ ^. 其实如果程序真的用了 **MD5** 之类的校验和,我们总是可以通过简单的检测程序来查看游戏的可执行文件是否使用了常见的校验/加密方式.这个留待以后的篇章再说吧.如果游戏既对归档文件使用了 **CRC32** 校验和而又用了 **PNG** 图像的话,那就有点不走运了...)

长话短说,上面所说的程序可以用.所幸的是没有什么诡异的校验和,归档的特征里也没什么我们没能再现的(说来,文件的顺序还是有可能有所谓的...)

下一步就是尝试修改点什么.不如试试修改标题菜单的背景图吧? 这幅图正好是 **x0000.png**,所以我们可以随便找张 **640x480** 的 **PNG** 图片替换掉它,重新打包,然后运行游戏看看...

那个高亮的长方形是主菜单的高亮系统的一部分.看来我们的进展很顺利...大部分的图形界面资源都是以 **PNG** 方式保存的,所以我们的翻译和改图员应该可以用对应工具来解决掉游戏的界面,没问题.

不过,等等...还有脚本呢!

```
% python crosschannel_extract2.py script.pd script
Expected "PackOnly" at position 0x0 but saw "PackPlus".
Aborting!
Traceback (most recent call last):
...
```

PackPlus?! 哦糟糕.不过我们的错误检查的高明之处终于展现了出来...我们马上就知道脚本归档里有诡异的地方了,而没有解出些垃圾等到后面的步骤才想办法弄清那些是什么.让我们回到十六进制编辑器来看看是怎么回事.

嗯,其实看起来挺正常的.那些零都还在;也有貌似文件数量,文件名,地址,大小等信息的数据.如果你一直向下滚动,直到 **0x240048** 之前你会看到都是 **0x00**,跟以前一样.那么整个文件头都跟之前的一样,除了特征标记之外.莫非我们做了无用功?

啊,还是看看原始文件本身:

这看起来有点怪.这些字节看起来不怎么随机——有长串的常量值——所以它应该没有被压缩过,也应该没被复杂的加密过.在这之中,有大量大于 **0x80** 的值,有点不寻常.

(译者按: 要吐槽了.有大量的大值确实不寻常,不过考虑到日文字符编码 **Shift-JIS(CP932)** 使用了 **C1** 控制码作为首字节,其实就算是正常的脚本里也很可能有大量大于 **0x80** 的字节...问题不在 **0x80** 上 **=_||** 问题是,如果一个脚本是没编译过的,其中的指令多数情况下用英文表示,编码会是 **ASCII** 兼容的,也就是说字节的值应该小于 **0x7F**,这就跟 **0x80** 扯上关系了.)

所以我们可以猜测所看到的是某种基本"加密".最常见的就是让数据与一个常量字节进行加(减)/异或,适合于保护文件不被你的六岁妹妹拆掉...假设她不会编程的话,呵呵.

我们完全可以直接猜出答案...事实上,如果你真的猜的话,说不定一下就猜到了.不过你知道吗,我很懒,所以我准备直接写些代码来暴力破解.

```
arcfile=open('script.pd', 'rb')
arcfile.seek(0x240076)
data=arcfile.read(16)
outfile=open('bruteforce.dat', 'wb')
for i in xrange(256) :
    for temp in data :
        outfile.write(chr(ord(temp) ^ i))
    for temp in data :
        outfile.write(chr((ord(temp) + i) & 0x00FF))
```



```
outfile.close()
arcfile.close()
```

我只是选了个看似有趣的地址读取了 **16** 个字节的数据(这个长度正好对应我用的十六进制编辑器的宽度),然后对这串数据与一个字节所能包含的所有可能值做异或和加操作,再把结果写到一个文件里方便查看.肉眼从头到尾扫一次这个 **8KB** 的文件大概要 **10** 秒,我们会发现:

哇,可辨认的文本,在文件的最最后...总是你最后才看到的地方.这里对应的是与 **0xFF** 做异或操作,也可以说是把文件的所有位都反转一次(**0** 与 **1** 互换,其实也就是做了一次非(**NOT**)操作);你很可能猜的就是这个.其实我想说的是,当你不确定的时候,不要怕做点小实验...写出上面的代码片段然后把一些可能性扫一次可能会花掉 **5** 分钟,到顶了;相反,如果你有时候不够聪明的话,很可能在一些简单却未知的加密上都会碰几小时甚至几天的墙.

现在我们可以更新一下我们的提取工具来对应这种加密.要修改处理文件头部分的代码很简单:

```
signature = arcfile.read(8)
if signature == 'PackOnly' :
    xorbyte = 0
elif signature == 'PackPlus' :
    xorbyte = 0x00FF
else :
    print 'Unknown file signature %s, aborting.' % \
        escape_string(signature)
    sys.exit(0)
```

同时修改处理文件本身的部分:

```
data = array('B', arcfile.read(size))
if xorbyte != 0 :
    for i in xrange(len(data)) :
        data[i] ^= xorbyte
outfile.write(data.tostring())
```

如果你不熟悉 **Python** 的话,这部分看起来可能会有点让你混乱.标准的 **Python** 字符串是不可修改的数据类型,所以你要想把所有字节都异或掉的话得创建那个字符串成千上万份的复本.与其那样,我们把数据读入一个可以修改的无符号字节数组.要这么做的话,你得在你的文件

顶上加上一行 `from array import array` 代码,因为数组模块不是 Python 语言的核心部分.新版本的提取程序是 `crosschannel-extract3.py`.

```
% python crosschannel-extract3.py script.pd script
Extracting 54 files...
Reading archive index...
Extracting adstart.dsf (236 bytes) from offset 0x240048
Extracting cca0001.dsf (934 bytes) from offset 0x240134
...
Extracting cca0011c.dsf (6674 bytes) from offset 0x297376
```

看来我们把问题解决了.解出来的这些文件能用文本编辑器以 `Shift-JIS` 编码正常打开.上下看看,我们可以发现 `cca0001.dsf` 文件是主要游戏脚本的开头.稍微编辑一下,快速重新打包(我们在自己打包的时候甚至不用理会 `PackPlus` 了),变!(呵呵玩魔术么...=_=)

注意到我跳过了最后这步里的一些麻烦的地方,例如说手动解决自动换行,还有确保游戏引擎能接收 `CR-LF` 换行而不是单个 `LF`.其实还有些诡异之处...我并不是说脚本格式完全不重要,只是,我们现在已经打开了超过这篇文章范围的实验的可能性.

从此,我们可以走向何方? 作为家庭作业,你应该用这段代码实验一下,如果可以自己用你选定的语言来写自己的代码更好.而且,归档里的那 `2.3MB` 的 `0x00` 该怎么办...如果你把那些零拿掉来节省空间的话,游戏引擎会抗议吗? 你能写个例程,在把脚本打包进归档时自动解决换行问题吗? 窗口标题栏里的汉字又怎么办? 你尝试做这个游戏的完整翻译时又会碰到什么问题呢?

不过,暂时来说,`Cross+Channel` 我只准备用到此为止...我们已经破解了相当大的一部分,而且从这里也没什么好再学的了.这是个合理的成就,因为这是个在市场上的实际游戏,而且人们会有兴趣翻译它.不过它也只是个简单的例子.这里的归档文件没有压缩,也只有轻微的加密;图像文件使用的是标准的格式;脚本是纯文本;而且游戏也没有完整性校验,我们甚至不需要去修改游戏的可执行文件.

(译者按: 把日语游戏英文化与中文化面对的难题不一样.对英文化工作者来说,他们常担心的是自动换行,还有显示半角字符的问题;而中文化工作者则得解决字符编码的不兼容,通常还是得修改游戏的可执行文件的.)

这些特征要是反过来的话,我们就得多做很多工作.所以,在接下来的几部分中,我会挑些别的游戏来作为更复杂的情况的好例子,每次集中关注一个方面.请自由推荐你关心的游戏,不过当然,我不保证那些游戏我都知道,呵.

===== 地味的分界线 =====

下面是对原文的部分回复,选取翻译,有修改.原文的所有回复,请到 **Part I** 前给出的原文地址查看.

Shish:

一个可预测,但值得注意的观察——这个游戏不在乎归档使用的是两种格式的哪一种.当加密是简单的异或时,这或许没什么用.但是当加密/解密方式很复杂时,把本来加密了的归档替换为无加密的就可能很有用.

Edward Keyes:

没错,这种事常有.游戏经常会有几种不同的读取数据的方式,特别是作为一种调试或者打补丁的机制,例如说"先看看文件是否就在磁盘上,没有的话再到归档里找找"之类.这些归档也常有未压缩的格式,为简单我们可以拿来一用...只要压缩标志是格式的一部分,游戏就能正确读取数据.最后,有时候你可以省下许多时间,用比较"蠢"的方式来实现一种图像格式或者一种压缩算法,例如:如果压缩算法允许用转义代码来向压缩流里插入原始字节的话,直接把所有字节都转义掉就可以了! (不过最终发布前还是应该回头好好实现一次的...)

在完整版的星之梦里,DRM 意味着我们要修改文件就得付出很大的代价.还好游戏有个后门,能让游戏引擎读取未加保护的版本.省了不少工夫,也可以说要不是这样整个翻译可能都不可能做出来,因为把 DRM 完全禁止掉可能给我们造成些道德问题.

Haeleth:

倒,Python.你应该用 OCaml 的.它有着 Python 所有的优点却没有 Python 的缺点(像驴一样慢,没有预先的类型检查,有大量的空白).

考虑到我所破解过的系统:Majiro 用了一种专有图像格式,与其说它复杂还不如说只是很丑陋(是个把 RLE 用到极限的傻系统,RLE = run length encoding,变长度编码).AVG32 有简单的压缩(而没有加密)和相对简易的字节码格式.而且你们也知道,RealLive 是个典型的例子,虽然有内建的工具但却破烂不堪,用它们的感觉简直就跟破解程序一样.

注意到 RealLive 的"外部"文件实际上是压缩过也轻微加密过的,因为 RealLive 是在原始文件级而不是归档级来处理压缩的.所以这么做(用不同版本的文件格式替换掉原来的)对 Kinetic Novel 可能有用,因为它在全局归档级上有隐秘的加密,不过对一般的 RealLive 游戏来说就没什么用了: 不管文件是否放进归档,游戏引擎都要你做压缩和加密的苦劳.

相似的,可以看看游戏在接受它们自己专有的格式之外是否也支持标准格式.例如,Majiro 似乎在用专有且让人不爽的 Rokucho 格式来储存图像,不过我没浪费一点时间去编写一个 Rokucho 压缩程序,因为游戏引擎也会高兴的接受 PNG 格式.虽然很多 RealLive 游戏坚持使用专有的 NWA 格式来储存音频,如果你想编辑或者替换掉那些音频的话,直接用 Ogg Vorbis 就可以了.

(译者按: Part III 完毕,为了保持原文的完整性,连原文的回复也选择翻译了上来,融合到原文中.是不是很丰富的一章呢? 希望大家都能从中获得一定经验,也请多多举出各种例子来分享经验.)

[So You Want To Be a Hacker? Part IV: Compression Formats](#)

Posted by Edward Keyes on July 27th 2006 to [Production Process](#), [So You Want To Be a Hacker?](#)

Game data archives almost always employ some type of compression for their contents, and sometimes will even mix and match different algorithms for different types of files. Understanding the typical compression formats is therefore crucial to the success of a game hacker.

Moreover, you need to be able to *recognize* the common algorithms just from their compressed data alone, so when you're staring at hex dumps, you will know how to proceed. In today's installment, we'll go through some of the most popular formats, how they work, and how you can recognize them "in the wild".

First of all, note that this task is in theory quite hard. The ideal compression algorithm produces data that is essentially indistinguishable from random noise (which counterintuitively contains the highest density of information). Fortunately, games have two additional requirements: to be able to access data quickly, and to be programmed by normal coders as opposed to Ph.D. computer scientists.

Both of these requirements means games typically use either industry-standard formats or relatively simple, quick algorithms that anyone can code from a book. The former often include extra identifying information we can spot, and the latter compress data poorly enough that it looks like "compressed data" instead of looking like random noise.

zlib

Let's start with the industry standard, [zlib](#). This is an open-source compression library which implements the classic 'deflate' algorithm used in .zip and .gz files. It's pretty fast and pretty decent, and since it's already written and completely free, it gets used all over the place, including in game archives.

How can you recognize it? **0x78 0x9C**. The first two bytes of a standard zlib-compressed chunk of data will be those two bytes, which specify the default settings of the algorithm ('deflate' with 32KB buffer, default compression level). Alternately, you will often see **0x78 0xDA**, which is the same except using the maximum compression level. If you see those two bytes at the start of where you expect a file to be, rejoice, since you've just solved a big mystery with next to zero effort.

Decoding this format is also pretty easy, since virtually every modern language will have a zlib library for it. In C, you want to just link against **libz** and call:

```
#include <zlib.h>
uncompress(new_buffer, &new_size, comp_buffer, comp_size);
```

Be sure to allocate enough memory for your expanded data: hopefully the archive index will have already provided you with the original file size. The function will return a status code and additionally update `new_size` with the amount of data that was uncompressed.

In Python, dealing with zlib is just embarrassingly easy:

```
new_data = comp_data.decode('zlib')
```

One of the built-in string-encoding methods (like ASCII, UTF8, Shift-JIS, etc.) is just zlib encoding, so if you have your data as a string you can just expand it like that. Alternately you can import zlib and use more direct function calls for extra control.

Compressing data is just as easy, with the `compress()` function in C — or `compress2()` if you want to specify the compression level — and the `encode('zlib')` string method in Python (or zlib library calls).

I don't want to say much about the inner workings of the deflate algorithm, since that really doesn't come up very often: you can safely treat it like a black box. However, there is one extra facet I've run across: the Adler32 checksum. This is a very simple 32-bit checksum algorithm (like CRC32) which is included in the zlib library, and therefore gets also used by games a bit. Additionally, the zlib format specifies that an Adler checksum is appended to the end of a compressed file for error-checking purposes.

However, some games twist their zlib implementation slightly by either leaving off the checksum (in favor of using their own elsewhere in the archive) or moving it into the archive index instead. This will cause the zlib `uncompress` call to return an error, *even though it actually uncompressed the data successfully*.

So, a word to the wise: if you're sure that the game is using zlib but you keep getting errors when you try to expand the data, look for this case. You may have to do a little twiddling of the compressed data to add the expected checksum at the end, or just ignore the zlib error codes and continue as normal.

Run-length compression

This is the simplest kind of home-grown compression you're likely to run across. It shows up in image compression a lot, sometimes as part of a larger sequence of processing. Basically the idea is to start with a sequence of bytes and chunk them up whenever you run across a repeated value:

```
31 92 24 24 24 24 C5 00 00
= 31 92 5*24 C5 2*00
```

Exactly how you represent the chunked-up data varies a bit from algorithm to algorithm, depending on what you expect the sequences to look like.

Escape byte. You might designate a byte, say **0xFF** as a flag for designating a run of repeated bytes, and follow it by a count and a value. So the above data would be:

```
31 92 FF 05 24 C5 FF 02 00
= 31 92 5*24 C5 2*00
```

If the flag byte actually appears in your data, you have to unescape it by, say, having a length of 0 in the next byte.

Repeated bytes. Here you just start running through your data normally, and whenever you have two bytes in a row that are the same, you replace all the rest of them (the third and thereafter) with a count byte:

```
31 92 24 24 03 C5 00 00 00
= 31 92 24 24 3*24 C5 00 00 0*00
```

If you don't have a third repeated value, you'll need to waste a byte to give a count of 0.

Alternating types. Here you assume that your data alternates between runs of raw values and runs of repeated bytes, and prepend length counts to each type:

```
02 31 92 05 24 01 C5 02 00
= 2 (31 92), 5*24, 1 (C5), 2*00
```

Naturally, if you have two repeated runs in a row, you'll have to waste a byte to insert a 0-length raw sequence between them. A special case of this I've run across is when you expect to have long runs of zero in

particular instead of any random byte, so you just alternate between runs of zeroes (with just a bare count value) and runs of raw data.

Note, of course, that there is some subtlety which can be involved depending on the variant you run across. For instance, it's often the case that pairs of bytes aren't efficient to encode, so they're just treated as raw data. Also, rather than giving lengths themselves, sometimes you encode, say, length-3, if length values of 0, 1, and 2 aren't ever needed. In some cases you might also run across multi-byte length values (controlled, say, by the high bit in the first length byte).

For images, you may have *pixels* instead of bytes which are the fundamental unit of repetition. In that case, even two RGB pixels in a row which are the same can be successfully compressed.

In any event, how do you recognize this format? The general principle is that all of these variations have to fall back on including raw bytes in the file a lot, so you want to try to look for those identifiable sequences (RGB triplets in image formats are good to key off of) interspersed with control codes. It's often helpful to have an uncompressed version of an image to compare against, which you can recover from a screenshot or from snooping the game's memory in a debugger (a topic for later articles).

LZSS

One step up from run-length encoding is to be able to do something useful with whole sequences of data that are repeated instead of single bytes. Here, the algorithm keeps track of the data it's already seen, and if some chunk is repeated, it just encodes a back-reference to that section of the file instead:

```
I love compression.  This is compressed!  
= I love compression.  This [is ][compress]ed!  
= I love compression.  This [3, -3][8, -22]ed!
```

The bracketed sections indicate runs of characters that have been seen before, so you just give a length and a backwards offset for where to copy them from. A lot of compression algorithms, zlib included, are based on this general principle, but one version that seems to crop up a lot is LZSS.

The special feature of this format is how it controls switching between raw bytes and back-references. It uses one bit in a control byte to determine this, often a 1 for a raw byte and a 0 for a back-reference

sequence. So one control byte will determine the interpretation of the next 8 pieces:

```
I love compression. This [3,-3][8,-22]ed!  
= FF "I love c" FF "ompressi" FF "on. Thi" 73 "s " 03 03 08 16 "ed!"
```

The **0xFF** control bytes just say “8 raw bytes follow”, and the **0x73** byte is binary **01110011**: reading from least-significant bit, that’s 2 raw bytes, 2 back-references, and then 3 raw bytes.

Recognizing this format in the wild rests on the control bytes, and you can spot it most easily in script files. If you see text which looks like this, **FF** with readable **tFF**ext plus some junk characters in every 9th byte, you’re dealing with LZSS. You can also spot this in image formats, since the natural rhythm of RGB triplets will get interrupted by the control bytes.

Note that the farther in the file you go, the harder this gets to recognize, since the proportion of back-references tends to climb once the algorithm has a larger dictionary of previously-seen data to draw upon.

The major hassle with this format is the nature of the back-references. There are a *lot* of subtle variants of this. One of the most popular ones uses a 4096-byte sliding window, and encodes back-references as a 12-bit offset in the window and a 4-bit length. However, is the length the real length or length-3? Is the offset relative to the current position, or is it an array offset in a separate 4096-byte ring buffer? Is the control byte read most- or least-significant bit first? I’ve even run across an example where there were several different back-reference formats: a 1-byte one for short runs in a small window, a 2-byte one for medium-length runs in a decent window, and a 3-byte one for large runs over a huge window. You will just need to experiment a little bit to see exactly what the particular game is doing, unfortunately.

One subtle point is that you may be allowed to specify a back-reference which overlaps with data you haven’t seen yet. By that I mean a length larger than the negative offset involved:

```
This is freeeeeeeeaky!  
= This [is ]fre[eeeeee]aky!  
= This [3,-3]fre[6,-1]aky!
```

The **[6,-1]** back-reference works because you are copying the bytes one at a time: first you copy the second ‘e’ from the first, and now you can copy the third ‘e’ from the second, etc. Be aware of this subtlety when

you implement your own algorithms, since (a) this can preclude you from doing certain types of memory copying or string slicing, (b) not all games will be able to understand this type of reference, so don't encode that way unless you know yours can.

Huffman encoding

From one point of view, this is easier than other algorithms since it only works on single bytes (or symbols, in general) at a time, but it's also more tricky since the compressed data is a bitstream rather than being easy-to-digest bytes and control codes.

It works by figuring out the frequencies of all the bytes in a file, and encoding the more common ones with fewer than 8 bits, and the less common ones with more than 8, so you end up with a smaller file on average. This is very closely related to concepts of entropy, since each symbol generally gets encoded with a number of bits equal to its own entropy (as determined by its frequency).

Let's be specific. Consider the string "abracadabra". The letter breakdown is:

a : 5/11 ~ 1.14 bits
b : 2/11 ~ 2.46 bits
c : 1/11 ~ 3.46 bits
d : 1/11 ~ 3.46 bits
r : 2/11 ~ 2.46 bits

Where I've given the number of bits of entropy each frequency corresponds to (i.e. if you have a 25% chance of having a certain letter, it has a 2-bit entropy since you need to give one of 4 values, say **00**, to specify it out of the other 75% of possibilities: **01**, **10**, **11**). Unfortunately we can't use fractional bits, so we may have to round up or down from these theoretical values.

How do we choose the right codes? Well, the best way is to build up a tree, starting from the least-likely values. That is, we treat, say, "c or d" as a single symbol with a frequency of 2/11, and say that if we get that far we know we can just spend one extra bit to figure out whether we mean c or d:

a : 5
0=c + 1=d : 2
b : 2

r : 2

Then we continue doing the same thing. At each step we combine the two least-weight items together, adding one bit to the front of their codes as we go. In the case of ties, we pick the ones with shorter already-assigned codes, or alphabetically first values:

a : 5

0=b + 1=r : 4

0=c + 1=d : 2

00=b + 01=r + 10=c + 11=d : 6

a : 5

000=b + 001=r + 010=c + 011=d + 1=a : 11

So the codes we end up with are:

a : 1

b : 000

c : 010

d : 011

r : 001

You will notice an excellent property of these codes: they are not ambiguous. That is, you don't have **1** for 'a' and **100** for 'b' ... as soon as you hit that first **1**, you know you can stop and go on to the next symbol without needing to read any more. Therefore, "abracadabra" just gets encoded as:

a	b	r	a	c	a	d	a	b	r	a
1	000	001	1	010	1	011	1	000	001	1
= 10000011 01010111 00000110										
= 83 57 06										

We've compressed 88 bits (11 bytes) down to 23 bits (just under 3 bytes). Almost always the bits are packed most- to least-significant in a byte.

One subtlety is the exact method of tree creation, which assigns the codes. The method described above is "canonical", but sometimes games will use their own idiosyncratic methods which you will have to match exactly to avoid getting garbage.

How do you recognize this in a data file? Well, the decompressor needs to know the codes, and the easiest way to specify this is to give it the

frequencies (or more easily, the bit weights) of the values so it can construct its own tree.

Therefore, the compressed data will usually start with, say, a 256-element table of bit weights. So if you see 256 bytes of **05 06 08 07 0C 0B 06** — values that are around 8 plus or minus a few — followed by horrendous random junk, you’re probably looking at Huffman encoding.

Sometimes instead of bit weights you’ll have the actual frequency counts instead, which might need to have a multi-byte encoding scheme if they’re above 256. In that case, you’re mainly looking for a few hundred bytes of “stuff” followed by a sharp transition to much more random data.

Other algorithms

Needless to say, the algorithms covered here are not the full range of compression formats out there. I’ll just briefly mention some others in case you run across them, though I haven’t really seen them in the wild.

Arithmetic encoding. This is vaguely related to Huffman encoding, in that you are working strictly with single bytes (or symbols) and trying to stuff the most frequent ones into fewer bits. However, instead of being restricted to an integral number of bits for each one, here you are allowed to be fractional on average.

This works by breaking up the numerical interval $[0,1)$ into subranges corresponding to each symbol: the more common symbols correspond to larger ranges, in proportion to their frequency. You start with $[0,1)$, and the first byte restricts you to the subrange for that symbol. Then the second byte restricts you to a sub-subrange, the third byte a sub-sub-range, etc. Your final encoded data is any single numerical value inside the tiny range you end up in: just pick the number in that range you can represent in the least number of bits as a binary fractional value.

Needless to say there are some good tricks for implementing this without using ludicrously-high-precision math, but I won’t go into that.

LZ77 (Lempel-Ziv ’ 77)

This is the core of the zlib deflate algorithm, but you’ll sometimes see variants outside of that standard, so it’s useful to know a little about. It’s basically a combination of standard back-references as in LZSS, plus Huffman encoding. You just treat the back-reference command “copy 8 bytes” as a special symbol, like a byte value of $256+8=264$.

Then, with this mix of raw data bytes and back-reference symbols, you run it through a Huffman encoding to get the final compressed output. Typically you will do something different with the back-reference offsets: either leave them as raw data, or encode them in their own separate Huffman table.

LZW (Lempel-Ziv-Welch)

When taught correctly, this is an algorithm with a mind-blowing twist at the end. As it runs through the file, it builds up an incremental dictionary of previously-seen strings and outputs codes corresponding to the dictionary entries. And then, at the end, when you start to wonder how to encode this big dictionary so the decompressor can use it to make sense of the codes, you just throw the dictionary away. Cute.

Of course it turns out that things are cleverly designed so that the decompressor can build up an identical dictionary as it goes along, so there's no problem. This algorithm was patent-encumbered for a while, so it didn't get as widely adopted as it might otherwise have been, but you might start seeing more of it these days.

Conclusion

I've focused here on lossless general-purpose compression: the sorts of things that are done to data at the archive level. There is also a stage below this, where data can be compressed before even being put into the archive: making raw images into JPEGs, PNGs, or other compressed image formats, and converting sounds to MP3s, OGGs, and so forth. In many cases those compression steps are just a lossy approximation to the original data, which is okay for graphics and sounds but bad for other files.

In a later installment, I'll be tackling image formats in particular in more detail, since you will tend to run across custom ones a lot, some of which include image-specific processing steps (like, say, subtracting pixels from their neighbors) which wouldn't make a lot of sense in a more general-purpose compression algorithm. Encryption is another later topic, since sometimes that will keep you from being able to recognize compressed data for what it is.

And naturally, if you've run across other general compression algorithms used in games you've looked at, please mention them in the comments, since I don't pretend to have investigated all the games out there... I'm still being surprised all the time.

12 Responses to “So You Want To Be a Hacker? Part IV: Compression Formats”

1. [Haeleth](#)

August 7th, 2006 | [6:02 pm](#)

Pretty comprehensive, though you forgot to mention at the start that this is where most people's brains explode. ^^

The only other common form of compression I've encountered in my own hacking is dictionary compression (trivial substitution of certain codes for a fixed set of particularly common strings, stored separately). And I've never seen that in a computer game, only in antiquated console titles, so it's not really worth covering in any depth.

2. September

August 31st, 2006 | [11:59 pm](#)

Thanks for the guide, it's been really helpful.

However, I'm at the aforementioned “brain explosion” stage trying to deal with (I think) an LZSS compressed file.

Some things came up that weren't mentioned in your guide... First of all, the control bytes don't always occur every 9th place, it seems to be taking 2 bytes for every back reference. I noticed the control bit is followed by 9 bytes instead of 8 if 7 of them are designated raw by the control bit.

Also, there's a 0x157F long section at the beginning of the file with lots of rather long and nearly identical repeating patterns. “BB AE B6 AB B0 B7 AC” honestly appears in this section like a hundred times.

Any further clarification on this compression format would help alot, or perhaps some direction to where I could get more info.

3. [Edward Keyes](#)

September 2nd, 2006 | [11:11 pm](#)

Yeah, the “every 9th byte” is only in the case where all 8 data bytes are raw instead of back-references, which is what you usually see at the very beginning of a file with a lot of unpredictable structure, like a script file. In the general case you can have a control byte after anywhere from 8 to 16 data bytes (or more, if the back references can span 3 bytes, though 2 is much more typical). The value of the control byte will tell you how many data bytes to expect.

I’ m not sure what to tell you about the repeating sequences. If it is LZSS, chances are the repeating sequences are a sign that the underlying file is also very repetitive, so you’ re getting the same back references over and over again. But my best advice there, if you have the option, is to look at some other files in the archive as well, rather than being solely focused on one: a lot of the time you’ ll find examples of files that compressed more or less well, giving you some extra examples to generalize from. If you’ re handy enough with a debugger to try to get access to the uncompressed data in the game’ s memory, that’ s also an awesome way to understand what’ s going on.

As for LZSS references, there’ s the Wikipedia entry of course, and you can search for tons of more info and code samples floating around the web, but the main trouble is that there isn’ t a single “LZSS format” like there is with zlib... I’ ve almost never seen exactly the same variant twice, as everybody has a different way to do the back references (the most common is 12-bit offset and 4-bit length, though that gets arranged in a few ways). So it’ s going to be difficult to find a reference that matches your particular variant exactly.

4. September

September 6th, 2006 | [11:37 pm](#)

I think I’ m beginning to see how this file is working, or at least how an LZSS file is supposed to work. One weird thing though is every other reference has really high values. Like a reference of 0xF5F0 at offset 0×47 in the file. Have you ever seen anything like this before? Maybe I’ m supposed to flip it or something :-/

sorry, I know this isn’ t a help thread. thanks for being helpful to a newbie.

5. [Edward Keyes](#)

September 7th, 2006 | [1:50 am](#)

Well, one thing you may need to work out is how the reference is encoding its values. Is 0xF5F0 a length of 0xF and an offset of 0×5F0, or a length of 0×0 and an offset of 0xF5F, or maybe if there's an endian swap going on (as you noted) it could really be 0xF0F5, with a similar breakdown of length and offset. Plus a length of 0xF may really mean, say, 17 or 18 bytes since lengths of 0, 1, or 2 might not ever be used, etc.

All the different possibilities get troublesome fast, so it's best to work with file examples where you can make decent guesses about the uncompressed data if possible. And of course if you're handy with a debugger you can grab the uncompressed data directly or just examine the uncompression assembly itself.

6. [September](#)

September 9th, 2006 | [1:31 am](#)

thanks for the info. I tried a couple debug programs and I'm honestly not even familiar enough with the terminology to understand what was going on.

but just in case you or anyone else was curious or has a similar experience in the future, what was happening to me is that the real offset was offset + 0×14. So if the real offset is less than 0×14 this gets wrapped back around. The control byte I was talking about, for example, [F5,F0] had an offset of 0xFF5 and a real offset of 0×9. Also, the offset is calculated from the start of the buffer, not spaces back. And the real length was length + 3.

man, that gave me headaches for days. but it really feels great to have figured it out. now I just need to learn how to code... :-P

7. [Edward Keyes](#)

September 9th, 2006 | [8:12 pm](#)

Congrats!

8. [Slow Fourier Transform » A reverse-engineering puzzle](#)

May 26th, 2007 | [2:50 pm](#)

[...] There's three compression types. 0×00 is raw storage, which was obvious. 0×01 is an LZSS flavor, which took me some 20 hours straight to decipher — it always starts with a control byte, stores references as AABC where BAA is the reference to a 0×1000 ring buffer (with a 0×12 offset for some silly reason) and C is the run length-3. Thankfully I had a set of files which were repackaged by a translator of the game who neglected to write a compression procedure, so I knew what to compare with and soon after I started writing actual code, it all came together, so I have extracted the majority of the content, including even a few forgotten PSD files. [...]

9. Criptych

May 28th, 2007 | [9:42 am](#)

I've seen a few of these formats before, especially the LZSS variants. Perhaps the strangest one I've come across (not counting SPB, used by NScripter) is the ZBM image format from the original X-Change; strange not because it's proprietary or complicated, but because you can decompress them with the Windows "expand" utility - it's the exact same format!

10. WinKiller Studio

June 9th, 2007 | [1:43 am](#)

ZBM is nothing but BMP with the first 100 bytes XORed with \$FF (compression is standard LWA or SZDD by Microsoft. They have used it for DOS and Windows 3.1 releases. It's really rare and even too old).

Currently i'm working on the real "universal" tool for the japanese visual novels fan-translators - Anime Editing Tools (available for downloading).

I want to decode Ever17 GCPS graphical data. It's some kind of zlib compression, but... my head is already blown to bits - help me if you can, guys!

11. WinKiller Studio

June 9th, 2007 | [1:50 am](#)

I' ve forgot to say this lately, but my tool is Open Source Delphi project.

So, i will appreciate any help (the author of IrfanView, Irfan Skiljan, is already helped me with RLE compression documentation).

More info on my site (the one english page there).

12. [WinKiller Studio](#)

June 11th, 2007 | [12:07 am](#)

I know, i' m a newbie (maybe even too much of spoiled newbie, forgive me. And forgive for ugly english too, ok?). Well, i' ve worked lately on decoding GCPS (or just CPS) graphical data format that is used in Ever17 – the out of infinity (© KID\HIRAMEKI Int.) (and, that' s my guess, in Memories Off 2nd too, coz they' re both built on the same engine. I even found PARTS OF FORGOTTEN Memories Off 2nd' s code right in the EXE! :)).

The format is really strange. **The game IS USING alpha channels.** But, when i' ve looked at the header...

Header structure (Object Pascal defs here):


```
<b>const 'CPS'+#0</b> – CPS header (4 bytes)<br>
<b>dword 4 bytes </b> – CPS file size<br>
<b>dword 4 bytes </b> – strange thing for sure! always equals to
16842854 or 0x66 0x00 0x01 0x01. Possibly a count of used colors
(16.8 million) or version number (11.102), or maybe a DOS date stamp
(01.08.1980, 00:03:12. WOW, if it's the really a date stamp, then
the PC that was used for compilation had problems with CMOS
battery...)<br>
<b>dword 4 bytes </b> – bitmap result stream size (not sure)<br>
<b>const 'bmp'+#0</b> – bmp header (4 bytes)
```

Come on, ask me “What' s so strange here and what you' re unsure of?”. I' ll tell you:

I' m using WinHex 10.4, so i was able to dump the loaded game data to hard drive. Then i' ve coded “GrapS” (now it' s a part of AnimED), the simple RAW scanner that is able to extract bitmap data from dump (with preview, sizing and jumping controls of course :))...

But, the thing i' ve never expected to see is that the size of resulting **alpha-channelled 32-bit bitmap** DOESN' T MATCH WITH CPS

header record (resampled 24-bit images is nearly matches, lacks of few (10-36) bytes).

I was (and still i am) confused and frustrated. Does that means that the game stores alphas in the separate files? The old games (such as Tokimeki Check-In! and X-Change 1-2) have used this trick, when the picture is divided into “the image” and “alpha” sections. But, not there! I’ ve estimated the size & other possible stuff... If alpha is stored in the same file, then the size MUST be the same as in the 32-bit bitmaps, because:

the size i’ ve calculated was $800 \times 600 \times (24 \div 8) = 1440000$ (raw, w/o header), but the “visible” part of bitmap is $800 \times 600 \times 24$ itself without the alpha, so there’ s definitely no space to store extra data.

Downsampled colors? No, i’ m an PC artist and would notice this trick very fast (believe me, i do).

Header lies. It only tells about the 24-bit image size, but excludes alpha channel? Or... what about those “extra” 36 bytes?

Alpha is stored elsewhere or combined from RGB channels on the fly? Hehe, not the second one, that’ s too complicated for sure.

Or, maybe the source is the 16-bit image with 8-bit alpha? I.e. A8R5G5B5. Non-standard, yes. And some “extra” byte here... but, poof! I’ ve checked several dumps – the count of used colors is more than 65535 (checked with IrfanView 4.00), so it’ s not possible...

So, where’ s the alpha is stored? Until i figure it out, i won’ t be able to write CPS (de)compressor, coz i’ m not a skilled programmer, only a designer-translator and can do nothing good with the bruteforcing and huffman-things. :(

Well, i’ m not exactly useless. Something is coming up to my mind. Every CPS file is ended with $0 \times 53 \ 0 \times 07$ or $0 \times 54 \ 0 \times 07$, and sometimes there a descending byte-sequences between the blocks. I think that the stream here should be readed reversible...

P.S. Thank you, Edward, for the very good and simple-to-understand articles (i will create the russian translation of it when i’ ll get your permission). STILL WAITING FOR THE PART V – “Graphical Data” . ^ _ ^

Maybe you’ ll show how to crack this tricky format... at least, i hope so...

Dmitri Poguliayev aka WinKiller Studio. Greetings from Russia, Kemerovo. A lot of Russian people loves anime. :)

P.P.S. About the game archives. There’ s something that bothers me. Did you’ ve seen the “Peach Princess” ARC files? I understand that the data optimization is something important, but... the filenames

and their extensions is stored *separately*. I think it's stupid, because it only saves a 262 kilobytes (12*65535 (filename with extension * maximum of possible file records) - 8*65535).

P.P.P.S. There is one really intriguing theme - Scripts, texts and Russian language.

Come to think of it, fans usually hacks japanese games to support latin, but where's the cyrillic characters support?! Ahh, it's not needed, right? NO, IT'S NEEDED. We are the normal humanoid creatures ;) , not aliens, and want to translate \play visual novels in our native language as well. I know the English, it's not the personally a problem of myself, but the others... I feel sad for them. :(

You're probably don't even know how it's hard to find a good japanese game outside of the Moscow. The most of buyable stuff is a pirated ripped releases (translated with Socrat, by the way) of the really old (1996-2004) h-games. I didn't even seen new titles such as X-Change 3 & X-Change Alternative. It's a surprise that the licensed US version of Ever17 was available not so long time ago (\$5 cost, that's really cheap, even for Kemerovo)...