BBFuzz:一种基于输入结构感知的协议模糊测试方案

翁嵩澠,贾　鹏,周安民
(四川大学网络空间安全学院,成都 610065)

摘　要:几乎所有需要通信的系统都离不开协议的设计,若协议栈存在漏洞,攻击者可以通过 Zero-Click 的方式达成拒绝服务攻击、信息窃取甚至是远程代码执行.协议消息具有一定的结构、语义、时序等要素,通用型模糊测试工具很难有效地对服务端进行模糊测试.近年来,有不少灰盒协议模糊测试的研究工作,其中比较具有代表性的工作是 AFLNET,然而这些研究工作对服务端状态机的覆盖依赖于初始种子集的覆盖面.本文首先分析了 AFLNET 无法完善处理二进制格式协议的缺陷,并提出了 BBFuzz,一款基于人工编写的数据模型进行测试用例生成的协议模糊测试工具.BBFuzz 能够在仅有一个初始输入的情况下,快速为种子队列提供众多感兴趣的种子文件,并且这些种子文件能够覆盖到较为全面的服务端状态.同时,BBFuzz 能够很好地支持两种不同类型的协议的模糊测试,即人类可读的 ASCII 格式和二进制格式的协议.本文实现了 BBFuzz 对 RTMP 协议的支持,并在两款知名的流媒体软件的 RTMP 模块上评估 BBFuzz.评估结果表明,BBFuzz 在 map density 和 paths 上的表现都优于 AFLNET.对于 RTMP 模块,本文在 ZLMediaKit 和 media-server 上分别挖掘到一个真实的漏洞,并且这两个漏洞都已经被分配了 HIGH 级别的 CVE 编号.

关键词:模糊测试;协议模糊测试;软件测试;协议安全
中图分类号:TP309.1　　**文献标志码**:A　　DOI:10.19907/j.0490-6756.2024.013002

BBFuzz: A protocol fuzzing tool combined with input structure-aware

WENG Song-Wei, *JIA Peng*, *ZHOU An-Min*
(School of Cyber Science and Engineering, Sichuan University, Chengdu 610065, China)

Abstract: Almost all of the systems which need communication are inseparable from protocol design. If the protocol stack is vulnerable, attackers can achieve denial of service attack, data theft and even remote code execution via Zero-Click. Protocol messages often have certain elements such as structure, semantics, and timing, making it challenging for general fuzzers to effectively perform fuzzing on the server. In recent years, there have been many researches on grey box protocol fuzzing, among which AFLNET is a representative one. However, the coverage of these researches on the server state machine depends on the coverage of the initial seed corpus. In this paper, we firstly analyze the defects of AFLNET in handling binary format protocols, and propose BBFuzz, a protocol fuzzer for test case generation based on manual data models. BBFuzz can quickly provide many interesting seed files for the seed queue, even with only one initial input, and these seed files can cover a more comprehensive server state. Meanwile, BBFuzz can well support fuzzing of two different types of protocols, namely human readable ASCII

收稿日期:2023-01-05
基金项目:国家重点研发计划项目(2021YFB3101803)
作者简介:翁嵩澠(1998－),男,硕士研究生,研究方向为二进制安全.E-mail:wengsongwei2019@163.com
通讯作者:贾鹏.E-mail:pengjia@scu.edu.cn

format and binary format protocols. The paper implemented BBFuzz's support for RTMP protocol, and evaluated BBFuzz on the RTMP module of two well-known streaming media software. Our evaluation results show that BBFuzz outperforms AFLNET on both map density and paths. For RTMP module, we dug two real vulnerabilities on ZLMediaKit and media-server respectively, and these two vulnerabilities have been assigned CVE number which is classified as HIGH.

Keywords: Fuzzing; Protocol fuzzing; Software testing; Protocol security

1 引　言

软件漏洞每年都会对政府、企业和研究机构等组织带来巨大的经济损失[1]. 若能在软件测试阶段,尽可能发现潜在的漏洞,就能大幅减少产品发布后由于漏洞带来的损失. 因此,软件测试技术是不可或缺的. 在软件的组件之中,协议的安全至关重要. 协议是通信双方必须共同遵从的一组约定,如消息的结构、含义、时序等. 桌面计算机、移动手机、物联网设备、工业控制系统和航空航天系统等,几乎所有需要通信的系统都离不开协议的设计(本文中的协议一般指计算机网络协议). 虽然协议的设计存在规范[2],但是由于各个开发商对于规范的解读、编码时的实现各不相同,因此服务端中可能存在各种各样的安全问题[3,4]. 并且由于服务端需要处理来自所有客户端的请求(当同处于一个网络环境下时),因此,服务端永远都暴露在可受攻击的环境下. 若服务端的协议栈存在漏洞,攻击者可以通过 Zero-Click 的方式达成拒绝服务攻击、信息窃取甚至是远程代码执行.

模糊测试是近几年最为流行的软件测试技术之一,在工业界和学术界被广泛使用,并挖掘了大量的漏洞证明其有效性,例如 afl[5], afl++[6], winnie[7], kafl[8]. 如今仍有许多研究人员致力于使模糊测试更快、更高效,例如 ijon[9], redqueen[10],然而他们主要针对具有通用文件型输入的测试目标. 协议模糊测试的主要目标为服务端,服务端往往具有状态机来表示其不同的状态,需要客户端通过消息序列来改变服务端的当前状态,以执行不同的处理逻辑. 另一方面,在数据交互过程中,有时客户端需要从服务端的响应包中提取数据,参与到后续的请求中. 这两个问题,对于测试目标为无状态、接收通用文件型输入的模糊测试工具而言,是很难处理的,这些模糊测试工具会产生大量在早期就被服务端丢弃的无效用例,并且难以推进服务端的当前状态.

研究人员提出了一些改进方案,例如编写一个 harness 来对测试目标进行指定状态下的单元测试[11],或者将消息序列拼接成一个文件作为输入[5]. 然而,这些方法都存在一定的弊端. 编写 harness 进行单元测试是使用通用模糊测试工具进行协议模糊测试的有效选择,然而一个 harness 所能覆盖的范围十分有限. 并且模糊测试的效果十分依赖于人工编写的 harness 的质量. 拼接消息序列并不是一个好的方案,因为协议的数据交互往往需要依赖时序.

Boofuzz[12] 仍然是如今比较流行的协议模糊测试工具. 其基于人工提供的协议状态模型,以及在某个状态下所接受的数据语法来生成消息序列. 尽管该工具的有效性非常依赖于开发者对于协议规范的理解所构造的状态模型和数据模型,不过该工具不需要获取测试目标的任何信息,因此在进行黑盒模糊测试时,其仍然是一个不错的选择. 然而,当能够进行灰盒模糊测试时,即使产生的种子文件是感兴趣的,例如发现了不在状态模型中的状态,该工具也不会保存任何信息. AFLNET[13] 指出了这个问题,并开发了一款能够自动更新服务端状态模型、通过状态覆盖和代码覆盖引导变异过程的模糊测试工具. 之后,一些研究人员基于 AFLNET 开发了更智能、更快速的模糊测试工具.

然而,这些工作均使用抓包的方式来获取初始种子输入. 当仅以一串消息序列作为模糊测试的初始输入时[14],很难产生另一个有效的输入. 例如,模糊测试很难从一个具有特定含义的字符串变异到另一个具有特定含义的字符串. 再比如,数据包是一个整体,里面各个字段相互联系,模糊测试很难从一个符合语法语义的数据包变异到另一个符合语法语义的数据包. 这个问题有一个缓解措施,就是提供海量的、状态覆盖全面的数据集,对其进行精简、裁剪后作为初始输入,这也正是我们在工程化模糊测试时所做的事. 当拥有覆盖面全的数据集后,变异过程有更大的概率产生新的感兴趣的种子文件. 然而,优质的数据集并不容易获得. 另一方面,即使我们拥有了海量的数据集,我们也不能确

保它的状态覆盖面是全面的，因为在协议规范中，有一部分，甚至是很大一部分的功能设计并不会在通常的通信过程中使用，而这一部分规范的具体实现往往可能造成漏洞.

我们提出了一款将基于数据模型生成测试用例，与通过字节级变异算法、消息级变异算法生成测试用例相结合的模糊测试工具 BBFuzz 来解决上述问题. 我们首先分析了对二进制格式协议和人类可读的 ASCII 格式协议进行模糊测试时存在的差异，若使用同一种方案来对待这两种不同形式的协议，正如 AFLNET 所做的，会对模糊测试的效果产生负面影响. BBFuzz 区别对待二进制格式协议和人类可读的 ASCII 格式协议，以使模糊测试能够更好地应用在各种目标协议上. 同时，BBFuzz 为模糊测试的变异过程引入了基于数据模型生成测试用例的方案，以使模糊测试进程能快速引入众多感兴趣的种子文件，即使仅以一串消息序列作为模糊测试的初始输入，也能快速地覆盖全面的状态. 在模糊测试的后期，可以在一个合理的时机关闭基于数据模型生成测试用例的方案，以使用随机性更强的字节级变异方案及消息级变异方案. 我们使用二进制格式流媒体协议 Real-Time Messaging Protocol(RTMP) 作为我们的测试目标，并在两款知名的流媒体软件上评估 BBFuzz：SRS[15] 和 ZLMediaKit[16]. 我们在 ZLMediaKit 和 media-server[17] 上的 RTMP 模块上分别挖掘到了一个被开发者认证的 Zero-day 漏洞，这两个漏洞均已被开发者修复，并且分别被分配了 CVE-2022-37237 和 CVE-2022-40016 的编号. 本文的实验结果表明，与 AFLNET 相比，BBFuzz 在覆盖率和种子队列的数量上有更好的表现.

本文的主要贡献有：(1) 将基于人工数据模型生成测试用例的方案集成到具有字节级和消息级变异方式的模糊测试进程中，并提出了 BBFuzz，一种能够在不具备良好构建的种子集的情况下，快速为模糊测试进程带来覆盖全面的、感兴趣的种子集的方法；(2) 分析了二进制格式协议和人类可读的 ASCII 格式协议的差异性，并且 BBFuzz 针对二进制格式协议进行了额外的处理，通过识别变异后的种子文件的边界来很好地处理各种各样的协议；(3) 实验结果表明，本文提出的 BBFuzz 在代码覆盖率和新增的种子数量上的表现都优于 AFLNET，且分别在两款知名的流媒体软件上挖掘到一个真实的漏洞，这两个漏洞均已被开发者认证和修复.

2　相关工作

当前，已经有许多被模糊测试社区广泛认可的灰盒模糊测试工具[5,6]，并且有许多研究人员致力于使灰盒模糊测试工具更快、更智能，例如 ijon[9]，redqueen[10]，Nagy 等人的工作[18] 和 weizz[19]，也有相关研究人员将灰盒模糊测试应用于不同的平台和目标，例如 winnie[7]，kafl[8]，firmafl[20]，然而，这些研究工作并不适用于我们的目标：协议. 一方面，服务端往往具有状态机模型，客户端需要向服务端发送请求以推进服务端的当前状态，否则输入会在早期就被服务端丢弃，而无法探测深层次的路径. 另一方面，协议往往具有一定的语法、语义规则，仅通过字节级的变异算法会产生大量无效的输入. 尽管使用这些通用文件型模糊测试工具仍然可对服务端进行模糊测试，但都存在一定的弊端，我们在第 1 节引言中指出了这些方法的问题.

近年来，有许多专门为协议模糊测试开发的工具，例如 aflnet[13]，stateafl[21]，snapfuzz[22] 和 snpsfuzzer[23]. 也有一些针对于特定协议开发的模糊测试工具，例如 ICS3Fuzzer[24] 和 TCP-Fuzz[25]. 在这些方案中，比较具有代表性的工具是 AFLNET. AFLNET 将消息序列作为种子文件，通过解析服务端的响应包来提取较为粗略的服务端状态的变化，记录种子文件中各个数据包导致的服务端状态变化，通过这种方法，在客户端构建服务端的状态机变化模型，同时将种子文件拆分为 M_1、M_2、M_3. 在模糊测试时，一次仅针对于一个服务端当前状态进行测试，这种做法是非常有效的. AFLNET 还在原有的字节级变异算法之上，添加了符合协议特性的消息级变异算法. 基于 AFLNET，Dongge Liu 等人[26] 使用了一种更为智能的状态选择算法. StateAFL[21] 使用插桩的方式来获取更为全面的服务端状态机表示. SnapFuzz[22] 使用快速同步通信、快照和重定向文件操作到定制的内存文件系统上的方案来提高模糊测试的速度. SNPSFuzzer[23] 设计了一种消息链分析算法来探索更多、更深的协议状态.

不过，由于协议具有一定的文法、语义，通过这些变异算法仍然难以覆盖到全面的服务端状态. AFLSmart[27] 将 Peach[28] 与模糊测试相结合，以使模糊测试工具对输入的结构感知. 这种方案与我们的思路类似，但 AFLSmart 仍然无法完成我们工作，因为它不是针对于协议设计的.

使用生成来进行模糊测试已有许多表现得很

好的方案,如 nautilus[29]、codealchemist[30]、squirrel[31]. 这些方案均采用将上下文无关语法转化为 AST(Abstract Syntax Tree),并在树上执行变异算法,然后将变异后的树转化为符合文法的测试用例的方法. 然而,已有工作表明,协议数据包并不适用于用上下文无关语法表示[32,33],对于协议数据包来说,使用数据模型来进行描述是我们所知的合适的方案.

尽管 Peach、Boofuzz 仍然是当前比较流行的基于生成的模糊测试方案,但是一方面他们是黑盒的,当我们进行灰盒模糊测试时,尽管生成的数据是感兴趣的,也不会让它参与到后续的模糊测试之中;另一方面,它们的设计更多的是一个工具而不是一个接口,将它们融入到其它工具之中都需要可观的工作量. 因此,我们使用 FormatFuzzer[34]作为我们的生成器,一款我们已知的最好的生成器接口. 对于不同的模糊测试工具,可以方便地使用同一个由 FormatFuzzer 生成的 so 文件将生成器集成到自己的工作中. 对于不同的格式,FormatFuzzer 仅需编写一份数据模型就可以同时完成解析和生成的工作,解析的结果可以参与到后续的模糊测试过程中. FormatFuzzer 的数据模型基于 010 Template[35]实现,对于不同的格式,编写数据模型所做的人工工作并不会太多,例如,对于 PNG 格式,仅需使用 493 行代码来实现[36].

3　协议的分类与差异

就文法而言,协议大体上可以分为两类:二进制格式和人类可读的 ASCII 格式[32]. 在 AFLNET 论文中,他们在 File Transfer Protocol (FTP)和 Real Time Streaming Protocol (RTSP)上进行了评估实验,这两个协议都是人类可读的 ASCII 格式的协议. 尽管 AFLNET 实现了对于几种二进制格式协议的模糊测试方案[37],如 TLS、SSH,但我们可以看到 AFLNET 使用同一种思路来处理两种不同类型协议的边界,我们随后将解释这样做的缺点. 在选定某种协议作为测试目标后,这两种不同类型的协议与模糊测试工作相关的差异就是其边界识别的不同.

以 RTSP 协议为例,在人类可读的 ASCII 格式中,存在诸如 "\r\n\r\n" 这样的终结符,而在二进制格式中,由字段来指示数据包的大小. 以 RTMP 协议为例,"fmt + chunk id" 字段指示了头部的大小,"body size" 字段指示了数据的长度. 对于符合文法、语义的数据包来说,要识别这两种类型数据包的边界都是容易且没有争议的. 对于人类可读的 ASCII 格式数据包,我们只需要一直读取缓冲区直到遇到终结符 "\r\n\r\n",在这之前的所有数据都可认定为是同一个数据包. 对于二进制格式数据包,我们只需要读取特定偏移的值,提取其语义后,就可以认定同一个数据包的范围,这也正是 AFLNET 所做的. 然而,在模糊测试的过程中,我们不可避免地会发现很多不符合文法、语义的数据包,这也正是我们所期望的. 当产生这些数据包后,对于人类可读的 ASCII 格式数据包,使用同样的方式来识别边界不会带来消极影响,如表 1 所示. 对于变异后的数据,同样可以很好地处理其边界. 在表 1~表 3 中,每个表中内容被分成三部分. 第一、三部分为原来的 region,第二部分为变异产生的新数据,同样的颜色在模糊测试过程中会被当成同一个数据包.

如果我们使用同样的方式来处理二进制格式数据包,那么其边界会被破坏掉,如表 2 所示. 由于每次都从特定的偏移读取数据并解析其语义,而模糊测试产生的随机数据会参与到这个识别过程中,造成边界的破坏. 我们不需要一直为服务端提供符合文法的数据,而应该把解析过程交给服务端来做,以期望探测到更多的漏洞. 我们可能更想要如表 3 所示的变异数据. 这种问题,在 resume 时尤其明显. AFLNET 使用同一个函数来处理添加队列、变异、重启时的输入. 由于添加队列时的初始输入是符合文法、语义的,因此不会带来任何问题;且变异时带来的影响正如我们分析表 2 时所描述的一致;而当重启时,此时的输入为变异后的数据包组合成的消息序列,AFLNET 把消息序列分成 M_1、M_2、M_3,M_1将服务端状态推进到我们所要的测试的状态上,M_2对当前状态执行模糊测试,M_3为后续状态的子消息序列. 由于 M_2经过变异,使用如表 2 所示的边界识别方案会带来边界的破坏,这种破坏会从 M_2开始,延续到 M_3中直到结束. 因此,此时消息序列带来的服务端状态变化和 resume 前模糊测试产生的结果不一致.

二进制格式数据包需要解析特定位置的语义来获取数据包的大小,由于模糊测试产生的测试用例是不可预期的,若还是采用这种方案,正如 AFLNET 所做的,那么无论如何我们都无法很好地识别变异后的数据的边界. 我们采用另一种方案来表示变异后的数据包的边界. 在读取初始输入时,我们同样使用解析函数(extract)来识别数据包的边界. 因为,此时的输入可以认为是符合文法、语

义的. 在重启模糊测试时，对于目录 queue 下的种子文件 A，我们不使用解析函数来识别其数据包的边界，而使用目录 replayable-queue 下的记录来识别数据包的边界. 因此，此时消息序列 A 中各个数据包的边界和先前模糊测试时的一致. 对于在测试过程中变异产生的 M_2，我们将其视为一整个块，而不调用解析函数(extract)将其拆分成多个数据包，通过这种方式，将解析操作转移到了服务端执行，以期望探测服务端预期外的解析导致的漏洞.

表 1　人类可读的 ASCII 协议变异后的数据包边界

Tab. 1　Message boundary after mutation in term of human-readable ASCII protocol

RTSP 数据包示例
DESCRIBE rtsp://127.0.0.1:8554/wavAudioTest RTSP/1.0\r\n CSeq: 2\r\n User-Agent: ./testRTSPClient (LIVE555 Streaming Media v2018.08.28)\r\n Accept: application/sdp\r\n \r\n
CCCCCCCCCCCCCCCC
SETUP rtsp://127.0.0.1:8554/wavAudioTest/track1 RTSP/1.0\r\n CSeq: 3\r\n User-Agent: ./testRTSPClient (LIVE555 Streaming Media v2018.08.28)\r\n Transport: RTP/AVP;unicast;client_port=47126-47127\r\n \r\n

表 2　二进制格式协议变异后边界破坏示例

Tab. 2　An example of broken boundary in term of binary protocol

RTMP 数据包示例	
0000	02 00 00 00 00 00 04 05 00 00 00 00 00 26 25 a0
0010	43 43 43 43 00 00 0a 43 43 43 43 43 43 43 43 43
0020	43 00 00 00 00 00 19 14 02 00 0c 63 72 65 61 74
0030	65 53 74 72 65 61 6d 00 40 00 00 00 00 00 00 00
0040	05

表 3　二进制格式协议变异后更合适的边界识别

Tab. 3　A preferable boundary after mutation in term of binary protocol

RTMP 数据包示例	
0000	02 00 00 00 00 00 04 05 00 00 00 00 00 26 25 a0
0010	43 43 43 43 00 00 0a 43 43 43 43 43 43 43 43 43
0020	43 00 00 00 00 00 19 14 02 00 0c 63 72 65 61 74
0030	65 53 74 72 65 61 6d 00 40 00 00 00 00 00 00 00
0040	05

4　设计与实现

我们为模糊测试的变异过程引入了基于人工提供的数据模型生成测试用例的方案，其架构如图 1 所示.

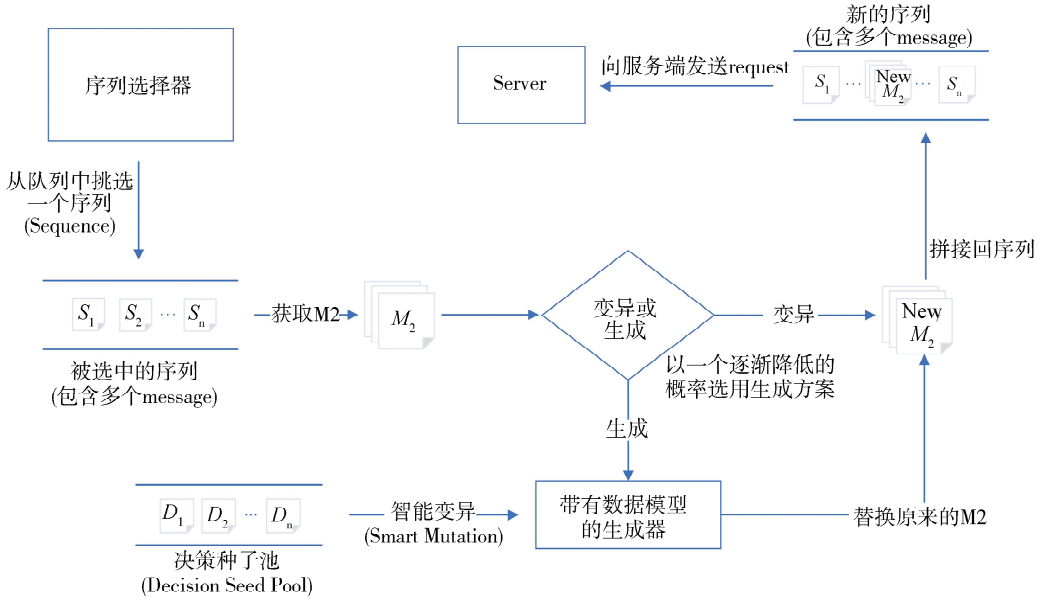

图 1　BBFuzz 架构图

Fig. 1　Architecture of BBFuzz

我们使用 FormatFuzzer[34] 作为 BBFuzz 的生成接口,我们在第 2 节中详细解释了我们选用该工具的原因. 当执行模糊测试时,以高概率选用变异方案,以相对较低的概率选用生成方案. 选用生成方案的概率会随着模糊测试的进行不断下降,并在一个合理的时间关闭生成方案. 虽然基于数据模型生成测试用例也可以探测目标潜在的漏洞,因为除了合法的值外,生成器同样会随机生成字段的值,但是在一定的生成次数之后,生成器的效果就会变得微乎其微. 在模糊测试的后期,随机变异(havoc)表现得更好,而在模糊测试的早期,生成器能快速地为模糊测试带来许多感兴趣的种子文件,并探测数据模型文法下的测试用例造成的漏洞. 在本节中,我们将详细描述生成器的实现细节.

4.1　数据模型

BBFuzz 使用 FormatFuzzer 作为生成接口,因此数据模型的描述方法与 FormatFuzzer 一致. 协议中存在一些无法简单生成的字段,常见的如 length、checksum、offset、compress、encode[33],对于其他容易生成的字段,我们仅需使用 010Editor Templates Language[35] 来描述它. 虽然 010Editor Templates Language 的本意是用于将二进制文件解析成层次型的结构(hierarchical structure),但是在 FormatFuzzer 中同样的描述形式也可用于生成测试用例. 对于无法简单生成的字段,FormatFuzzer 作出了相应的扩展,我们将在本节中简要地描述这些字段的描述方法.

(1) Length. Length 字段直到数据生成前都是无法确定它的值的,因此一般采用先生成数据再返回来确定 Length 字段的值的方案,即使是在其他较简单的非协议数据生成的工具中,也能较为容易地描述该字段. 表 4 展示了描述 Length 字段的方案,我们先记录生成的数据块的大小,随后回到先前用于占位但不具有正确的值的 Length 字段的位置,将 Length 字段修改为正确的值.

表 4　数据模型中长度字段的描述

Tab. 4　Data model of length field in BBFuzz

长度字段的描述算法

```
local uint32 pos_start = FTell();
CHUNKDATA chunkdata;
local uint32 pos_end = FTell();

local uint32 correctLength = pos_end-pos_start;

FSeek(LengthField);
// cover original hint length field with correct value
uint32 length = { correctLength };
FSeek(pos_end);
```

(2) Constraints. 在协议数据包中,经常会出现字段 y 依赖于字段 x 的情况. 例如 type 字段的值决定了 body 的结构. 当 x 在 y 之前生成时,这种情况比较容易处理. 我们只需要先生成 x,再根据 x 的值来决定 y 的生成即可. 然而,当 x 在 y 之后生成时,我们需要先固定 x 的值,再根据 x 的值来决定 y 的生成. 如表 5 所示,我们先用前瞻函数 ReadByte(s)来决定 x 的值,一旦确定了 x 的值,在后续生成 x 时就会使用这个确定的值. 由于 x 的值已经提前确定了,那么就可以根据 x 的值来决定 y 的生成.

表 5　数据模型中约束的描述

Tab. 5　Data model of constraint in BBFuzz

约束字段的描述算法

```
// it is the same when len(x) > 1
local byte x[] = { value01, value02, value03 ... }
switch(ReadByte(xposition,x)){
  case value01:
    ystruct01
  break;
  case value02:
    ystruct02
    break;
  ...
}
...
xposition: byte x;
```

(3) Chunk Ordering. 在某些文件结构中,可能存在上下文敏感的块顺序. 例如在 PNG 格式中,IHDR 必须是文件的第一个块,IEND 必须是文件的最后一个块,有些可选块必须位于 IDAT 块之前,有些可选块则没有限制. 不过,协议数据包通常不会存在上下文敏感的块顺序,更多时候,我们可能想从多种块中随机挑选一部分块,并且允许重复块的出现以探测潜在的漏洞,而不考虑块之间的顺序关系.

表 6 展示了从多种块中随机挑选一部分块,并且允许重复块的出现的方案. 我们重复从块列表中进行挑选,并以给定的概率挑选退出标识.

(4) Compress and Encode. 在协议数据包中,对某些域进行压缩和编码是很常见的. 由于压缩和编码具有不同的算法,因此在 BBFuzz 中通过自定义的函数来处理压缩和编码,对于常见的算法则可以调用相应的函数接口.

通过使用本节中所展示的数据模型的描述方法,我们可以很好地表示协议中的复杂字段,并生成符合文法的协议数据包.

表 6　数据模型中块的随机挑选描述

Tab. 6　Random choose chunk(s) in BBFuzz

块的随机挑选描述算法

```
local string blocknamelist[] = { "bk01", "bk02" ... };
local int continueRun = 1;
while(continueRun){
  switch(RandomChoose(blocknamelist, exitprobability)){
  case "bk01":
    BK01 block01;
    break;
  case "bk02":
    BK02 block02;
    break;
  ...
  case "mEND": // if exit is chosen, return an exit ID
  default:
    continueRun= 0;
    break;
  }
}
```

4.2　决策种子池

我们使用智能变异(Smart Mutation)作为BBFuzz的生成器，因为该方案在Dutra等人[34]的实验中表现最好，我们将在下一节简要描述该算法. 通过我们提供的数据模型，生成器能将数据包解析成Decision Seed，该过程类似于序列化，Decision Seed与数据包能够相互转化. 在Smart Mutation中，会将各个测试用例的Decision Seed保存起来，称为决策种子池(Decision Pool)，从池中随机挑选Decision Seed(s)执行智能变异算法，如抽象、替换、插入、删除等，即可产生新的测试用例.

在协议模糊测试工具中[13,21]，我们使用消息序列(包含多个数据包)作为种子输入，而对于生成器而言，同样需要将种子文件解析成Decision Seed并放入池中，换句话说，生成器需要Decision Seed与种子文件的一一对应. 然而此时，种子队列保存的内容为消息序列，而生成器的对象为单个数据包，两者产生了冲突. 因此，我们为BBFuzz创造了第二个队列，专门用于保存由生成器生成的M_2数据包，此时BBFuzz的种子队列如图2所示.

当模糊测试产生感兴趣的种子文件时，会将种子文件，即消息序列添加到队列中，这是模糊测试的主队列. 当感兴趣的种子文件的M_2是由生成器产生时，同时会将该M_2添加到FORMAT QUEUE中，并由生成器解析其相应的Decision Seed，FORMAT QUEUE及DECISION POOL供生成器使用，而QUEUE的含义与AFL保持一致.

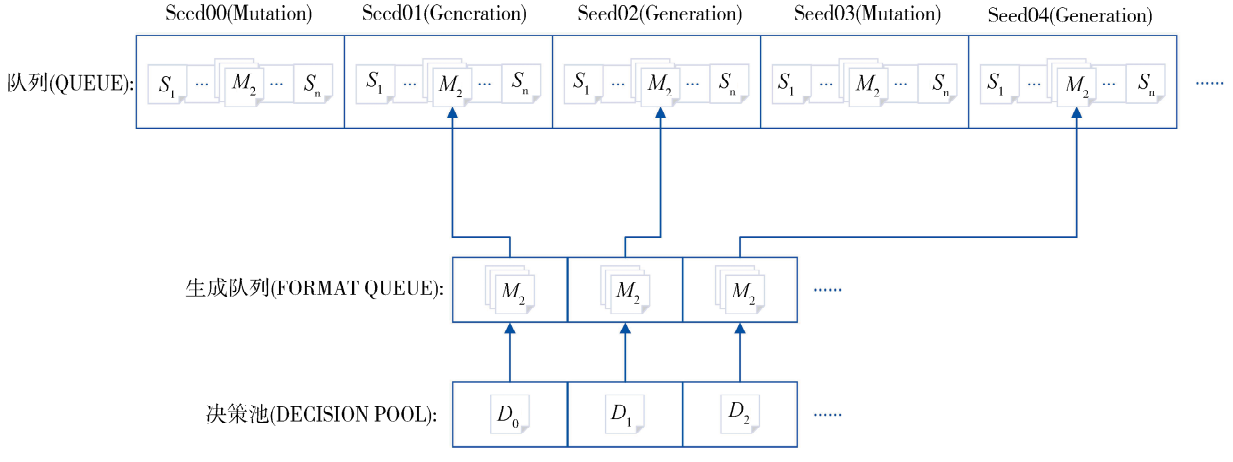

图 2　BBFuzz 中的种子队列

Fig. 2　Queue in BBFuzz

在FormatFuzzer中，决策种子与队列种子文件一一对应，这种形式表明，对于模糊测试变异产生的不符合文法的种子文件，也对其进行了解析，并将其Decision Seed放入池中. 然而，针对这些种子文件进行解析的有效性(validity)往往很低. 在BBFuzz中，我们仅对生成器产生的用例进行解析而不解析变异产生的用例，因为我们的主要目的是迅速为种子队列提供许多感兴趣的种子文件，以覆盖众多状态，而不是将生成器当成主要的模糊测试工具.

4.3　生成算法

生成器使用如下几种算法来产生新的测试用例：

(1)Smart Abstraction. 对于决策种子A，我们随机选定A中的某一个块c进行替换，复制块c之前的决策，并随机产生一串decision bytes来替换块c的决策，紧接着继续复制块c之后的决策.

(2) Smart Replacement. 对于决策种子A中

的某一块 c_1，我们挑选另一个决策种子 B 中的某一块 c_2，复制 c_1 之前的决策，并使用 c_2 的决策替换 c_1 的决策，紧接着继续复制 c_1 之后的决策.

(3) Smart Deletions/Insertions. 插入和删除功能仅针对于可选块进行. 生成器中可选块的含义就像 PNG 格式中的 bKGD 块，这些块是可选的，但是可能需要符合一些位置的限制. 在协议数据包中，由于通常不会存在上下文敏感的块顺序，因此我们通常不会采用这种描述形式. 表 6 中所展示的描述形式也可以达到删除/插入的效果.

5　实验设计与分析评估

我们将 AFLNET 作为我们实验的基准，并在两款知名的开源流媒体软件，srs[15] 和 ZLMediaKit[16] 上评估 BBFuzz 的有效性. 我们选用 RTMP 协议[38] 作为我们的测试目标，该协议在包含有直播功能的流媒体软件中广泛使用. 我们首先扩展 AFLNET 以使其支持 RTMP 协议，该协议在 AFLNET 中不被支持. 其次，我们为生成器编写了一份 RTMP 协议的数据描述模型，该数据模型被称作 Template. 我们为 BBFuzz 和 AFLNET 提供同一份初始输入，该初始输入通过抓包获取[14]，启动服务器并推流后，使用 vlc media player[39] 拉流以获取初始消息序列. 我们为 BBFuzz 提供 RTMP Template 而不为 AFLNET 提供，因为 AFLNET 不具备基于数据模型生成的功能. 我们分别对 srs 和 ZLMediaKit 的 RTMP 模块进行 3 次模糊测试，取实验的平均值作为参考以缓解随机性. 在模糊测试过程中，我们同时对测试服务端进行推流，以探索更多的服务端行为. 我们为 srs 的模糊测试设置 17 的 MAP_SIZE_POW2，为 ZLMediaKit 的模糊测试设置 18 的 MAP_SIZE_POW2，因为 ZLMediaKit 的路径多，而过高的 map density 会造成很多的路径碰撞. 我们取 map density 和种子文件的数量作为指标来评估两款工具的表现，并针对实验的结果进行分析. 我们的实验结果表明，BBFuzz 在代码覆盖率和新增的种子数量上的表现都优于 AFLNET，并且分别在两款知名的流媒体软件上挖掘到一个真实的漏洞，这两个漏洞均已被开发者认证和修复. 通过这种实验结果，可以验证 BBFuzz 能够在不具备良好构建的种子集的情况下，快速为模糊测试进程带来覆盖全面的、感兴趣的种子集.

5.1　代码覆盖率和新增的种子文件数量对比

我们在第一次 fuzz_one 执行后记录 0 h 时的状态，因此 0 h 具有初始的值. 图 3 展示了，BBFuzz 在整个模糊测试过程中，都表现得比 AFLNET 好. AFLNET 在 10 h 时探索的路径与 BBFuzz 在 4 h 时探索的路径相当. BBFuzz 在模糊测试时的种子文件数量也一直比 AFLNET 多，这正如我们所期望的，因为 BBFuzz 会为模糊测试提供许多的感兴趣的、符合文法的种子文件，而不仅仅是靠随机变异来产生. 然而，在图 4 中，我们可以看到，

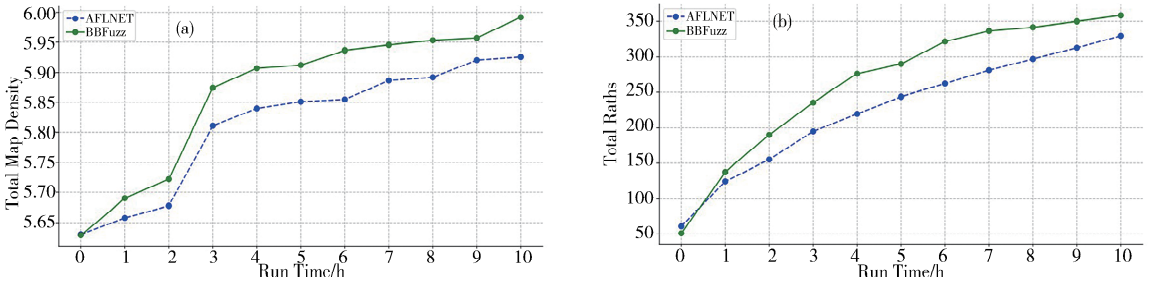

图 3　BBFuzz 与 AFLNET 在代码覆盖率和新增的种子文件数量上的对比(srs)

Fig. 3　The comparison of BBFuzz and AFLNET in map density and paths (srs)

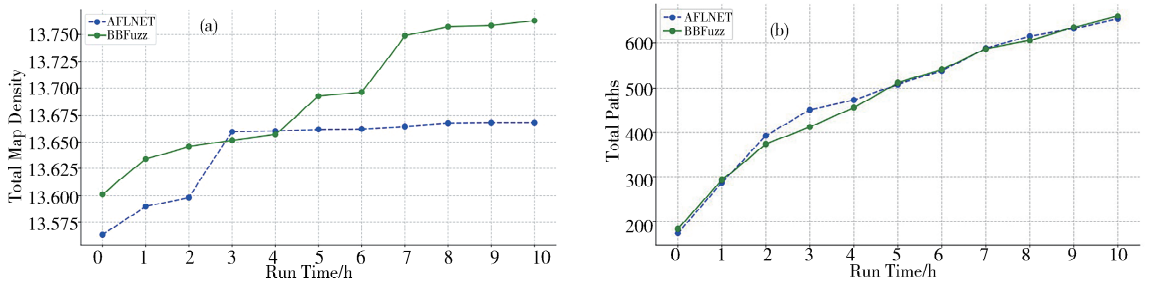

图 4　BBFuzz 与 AFLNET 在代码覆盖率和新增的种子文件数量上的对比(ZLMediaKit)

Fig. 4　The comparison of BBFuzz and AFLNET in map density and paths (ZLMediaKit)

BBFuzz 的种子文件数量一直和 AFLNET 的种子文件数量差不多. 我们分析了这种结果：在对 ZLMediaKit 进行模糊测试时，其稳定性并不高，这说明了 ZLMediaKit 存在很多与测试模块无关的代码. 尽管我们已经在原生的 AFLNET 基础上，添加了额外的同步机制，这也导致了我们实验时的速度比原生的 AFLNET 低，但是在一段时间的模糊测试后，其稳定性必然会下降到一个趋近稳定的值，这也说明了这些代码确实是和测试模块无关的. 而当稳定性并不高时，模糊测试会将很多并不感兴趣的种子文件当成感兴趣的种子文件添加到队列里，造成种子队列数量的虚高. 从代码覆盖率的变化，我们可以看出这一点，很显然，差不多的种子文件数量，其得到的总的代码覆盖率却有很大的差异，因为 BBFuzz 的种子文件队列中，具备更多的真正感兴趣的种子文件. 图 4 展示了 AFLNET 在 10 h 时探索的路径与 BBFuzz 在 4 h 时探索的路径相当，而且在 3 h 以后，AFLNET 所探索的路径并没有发生明显的变化，因为仅使用变异很难产生新的感兴趣的种子文件. 相比之下，BBFuzz 所探索的路径一直在稳步增长. 在模糊测试 2～3 h 时，AFLNET 的模糊测试有一个比较大的拐点，这说明对于 ZLMediaKit 来说，模糊测试可以比较稳定地探测到 3 h 时所探测到的路径.

5.2　漏洞的发现

我们分别在两款知名的流媒体软件中挖掘到一个真实的漏洞，包括 ZLMediaKit 和 media-server[17]. 在模糊测试 ZLMediaKit 时，种子文件会使服务端设置一个过小的 window size，这种行为会导致服务端发送 sendAcknowledgement 时无限递归，占满栈空间，最后导致崩溃. 该漏洞很容易通过生成器产生的数据探索到. 在模糊测试 media-server 时，种子文件在一个 session 内向服务端发送两次 PLAY 命令会引起服务端预期外的处理，最终导致 UAF 的发生. 该漏洞可以通过生成器的重复，或者消息级变异算法探索到，而对于不支持消息序列的模糊测试工具，则很难探测到该漏洞.

6　结　论

由于协议消息具有一定的结构、语义、时序等要素，使用通用文件型模糊测试工具的改进方案来进行协议模糊测试仍然存在一些弊端，而现有的灰盒协议模糊测试研究工作对服务端状态机的覆盖依赖于初始种子集的覆盖面. 基于这些问题，我们提出了 BBFuzz，一款将基于人工编写的数据模型进行测试用例生成与模糊测试进程相结合的协议模糊测试工具. BBFuzz 能够在仅有一个初始输入的情况下，快速为种子队列提供众多感兴趣的种子文件，并且这些种子文件能够覆盖到较为全面的服务端状态. 我们在两款知名的流媒体软件上评估 BBFuzz. BBFuzz 在 map density 和 paths 上的表现都优于一款已经存在的、具有代表性的协议模糊测试工具. BBFuzz 挖掘到了两个真实的漏洞. 现如今，物联网设备增长迅速，其安全问题也备受关注. 我们注意到，在物联网设备中协议被大量地使用，并且其中还有不少的私有协议. 接下来，我们计划拓展 BBFuzz 以支持对物联网设备协议安全的测试.

参考文献：

[1] Skyboxsecurity. Vulnerability and threat trends report 2022[EB/OL]. [2022-12-22]. https://www.skyboxsecurity.com/wp-content/uploads/2022/04/skyboxsecurity-vulnerability-threat-trends-report-2022_041122.pdf.

[2] Schulzrinne, Columbia. {RFC2326}: Real time streaming protocol [EB/OL]. [2022-12-22]. https://www.rfc-editor.org/rfc/rfc2326.html.

[3] Mitre. CVE-2019-7314[EB/OL]. [2022-12-22]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7314.

[4] Mitre. CVE-2019-15232[EB/OL]. [2022-12-22]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15232.

[5] Michal Zalewski(lcamtuf). American fuzzy lop[EB/OL]. [2022-12-22]. https://github.com/google/AFL.

[6] Fioraldi A, Maier D, Heuse M, *et al*. AFL++: combining incremental steps of fuzzing research [C]//Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20). Online: USENIX Association, 2020: 10.

[7] Jung J, Tong S, Hu H, *et al*. Winnie: fuzzing windows applications with harness synthesis and fast cloning [C]//Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021). Online: ISOC, 2021: 24334.

[8] Schumilo S, Aschermann C, Gawlik R, *et al*. kAFL: hardware-assisted feedback fuzzing for OS kernels [C]//Proceedings of the 26th USENIX Security Symposium (USENIX Security 17). Vancou-

ver, Canada: USENIX Association, 2017: 167.

[9] Aschermann C, Schumilo S, Abbasi A, *et al*. Ijon: exploring deep state spaces via fuzzing[C]//Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP). San Francisco, USA: IEEE, 2020: 1597.

[10] Aschermann C, Schumilo S, Blazytko T, *et al*. REDQUEEN: fuzzing with input-to-state correspondence[C]//Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS 2019). San Diego, USA: ISOC, 2019: 23371.

[11] LLVM organization. libFuzzer-a library for coverage-guided fuzz testing [EB/OL]. [2022-12-22]. https://llvm.org/docs/LibFuzzer.html.

[12] Jtpereyda. Boofuzz: network protocol fuzzing for humans[EB/OL]. [2022-12-22]. https://github.com/jtpereyda/boofuzz.

[13] Pham V, Böhme M, Roychoudhury A. AFLNet: a greybox fuzzer for network protocols[C]//Proceedings of the 13th International Conference on Software Testing, Validation and Verification (ICST). Porto, Portugal: IEEE, 2020: 2159.

[14] Thuan P, Max M, Paul B, *et al*. Prepare message sequences as seed inputs[EB/OL]. [2022-12-22]. https://github.com/aflnet/aflnet # step-1-prepare-message-sequences-as-seed-inputs.

[15] Lin W, Hong X Z, Lin C D, *et al*. Simple realtime server[EB/OL]. [2022-12-22]. https://github.com/ossrs/srs.

[16] Xia C. ZLMediaKit[EB/OL]. [2022-12-22]. https://github.com/ZLMediaKit/ZLMediaKit.

[17] Chen I. Media-server [EB/OL]. [2022-12-22]. https://github.com/ireader/media-server.

[18] Nagy S, Hicks M. Full-speed fuzzing: reducing fuzzing overhead through coverage-guided tracing [C]//Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP). San Francisco, USA: IEEE, 2019: 787.

[19] Fioraldi A, Daniele C, Coppa E. WEIZZ: automatic grey-box fuzzing for structured binary formats [C]//Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. Online: ACM, 2020: 1.

[20] Zheng Y, Davanian A, Yin H, *et al*. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation [C]//Proceedings of the 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, USA: USENIX Association, 2019: 1099.

[21] Natella R. StateAFL: greybox fuzzing for stateful network servers [J]. Empir Softw Eng, 2022: 27.

[22] Andronidis A, Cadar C. SnapFuzz: an efficient fuzzing framework for network applications[C]//Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2022). Online: ACM, 2022: 340.

[23] Li J, Li S, Sun G, *et al*. SNPSFuzzer: a fast grey-box fuzzer for stateful network protocols using snapshots[J]. IEEE T Inf Foren Sec, 2022, 17: 2673.

[24] Fang D, Song Z, Guan L, *et al*. ICS3Fuzzer: a framework for discovering protocol implementation bugs in ICS supervisory software by fuzzing [C]//Annual Computer Security Applications Conference 2021. Online: IEEE, 2021: 849.

[25] Zou Y, Bai J, Zhou J, *et al*. TCP-Fuzz: detecting memory and semantic bugs in TCP stacks with fuzzing [C]//2021 USENIX Annual Technical Conference. Online: USENIX Association, 2021: 489.

[26] Liu D, Pham V, Ernst G, *et al*. State selection algorithms and their impact on the performance of stateful network protocol fuzzing[C]//Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER). Honolulu, USA: IEEE, 2022: 720.

[27] Pham V, Marcel B, Santosa A, *et al*. Smart greybox fuzzing [J]. IEEE T Software Eng, 2019, 47: 1980.

[28] Michael E. Peach fuzzing platform[EB/OL]. [2022-12-22]. https://gitlab.com/peachtech/peach-fuzzer-community.

[29] Aschermann C, Frassetto T, Holz T, *et al*. NAUTILUS: Fishing for deep bugs with grammars [C]//Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS 2019). San Diego, USA: ISOC, 2019: 23412.

[30] Han H, Oh D, Cha S. CodeAlchemist: semantics-aware code generation to find vulnerabilities in javascript engines[C]//Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS 2019). San Diego, USA: ISOC, 2019: 23263.

[31] Zhong R, Chen Y, Hu H, *et al*. Squirrel: testing database management systems with language validity and coverage feedback[C]//Proceedings of the 2020 ACM SIGSAC Conference on Computer and Com-

munications Security. Online：ACM，2020：955.

[32] Pang R，Paxson V，Sommer R，*et al*. Binpac：a yacc for writing application protocol parsers[C]//Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement 2006. Rio de Janeriro，Brazil：ACM，2006：289.

[33] Bangert J，Zeldovich N. Nail：a practical tool for parsingand generating data formats[C]//Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation（OSDI 14）. Broomfield，USA：USENIX Association，2014：615.

[34] Dutra R，Gopinath R，Zeller A. FormatFuzzer：effective fuzzing of binary file formats[EB/OL]. [2023-10-29]. https://dl. acm. org/doi/10. 1145/3628157.

[35] Sweetscape Software. 010 Editor templates language reference[EB/OL]. [2022-12-22]. https://www. sweetscape. com/010editor/manual/TemplateVariables. htm.

[36] Dutra R，Gopinath R，Zeller A. PNG template example[EB/OL]. [2022-12-22]. https://github. com/uds-se/FormatFuzzer/blob/master/templates/png. bt.

[37] Pham V，Böhme M，Roychoudhury A. Aflnet header file[EB/OL]. [2022-12-22]. https://github. com/aflnet/aflnet/blob/master/aflnet. h.

[38] Hardeep P，Michael T. RTMP protocol specification[EB/OL]. [2022-12-22]. https://rtmp. veriskope. com/docs/spec/.

[39] Billy B，Sven H，Samuel H，*et al*. VLC media player[EB/OL]. [2022-12-22]. https://www. videolan. org/.